Horizontal Fusion Developers Guidance

HORIZONTAL FUSION: CATALYST FOR NET-CENTRIC TRANSFORMATION · DEPARTMENT OF DEFENSE

Department of Defense Assistant Secretary of Defense for Networks and Information Integration/DoD CIO

18 November 2004

HORIZONTAL FUSION

Revision Sheet

Release No.	Date	Revision Description
1.0	11/18/2004	Developers Guidance - Baseline

TABLE OF CONTENTS

TABLE OF FIGURES

1 INTRODUCTION

Horizontal Fusion was established by the Assistant Secretary of Defense for Networks and Information Integration (ASD/NII) in January 2003 to explore and investigate the potential feasibility of implementing an interoperable Net-Centric services-oriented architecture (SOA) environment for the Department of Defense (DoD). The introduction of a Net-Centric environment and SOA for the Department will deliver significant improvements in battlefield operations, including near-real time access to critical information, enhanced situational awareness (both on the battlefield and globally), informed and expeditious command decision-making, and controlled and coordinated operational and tactical military movements.

In addition, Horizontal Fusion was directed to implement a new approach to information technology (IT) management within the Department—Portfolio Management. Portfolio Management marks a significant change from existing management practices, under which purchasing decisions are made on a program-by-program basis. The Portfolio Management approach, according to the "Information Technology Portfolio Management" policy directive of 22 March 2004 issued by Deputy Defense Secretary Paul Wolfowitz, will ensure that the military possesses the "right capabilities to perform its mission and conduct effective operations, eliminate outdated ways of doing business, and achieve DoD's Net-Centricity goals." All DoD IT investments and programs will be managed as portfolios in the near future.

Horizontal Fusion is a direct response to operationalize, integrate, and optimize technology and operations to achieve "Power to the Edge" in the new battlespace. Horizontal Fusion is made possible by the new technology context that includes Wideband Satellite Communications (SATCOM), Global Information Grid Bandwidth Expansion (GIG-BE), and the Joint Tactical Radio Systems (JTRS). The increased bandwidth these capabilities will provide will deliver significant improvements in battlefield operations including real-time access to critical knowledge and enhanced situational awareness, informed and expeditious command decision-making, and operational and tactical response.

The Horizontal Fusion Portfolio Initiative is a management process with an emphasis on outcome-based performance intended to accelerate the transition of Net-Centric warfighting capabilities into the operational inventory.

The Horizontal Fusion Portfolio is a set of technologies and processes that have been selected to demonstrate their interoperability with other Initiatives of the Portfolio and their contribution to the overall concepts of Net-Centricity.

The intent is to accelerate the transition of Net-Centric warfighting from vision to reality. The Portfolio Initiative emphasizes outcome-based performance. Goals are closely aligned with the DoD vision of Transformation to Net-Centricity, with the objective of building practical solutions. The Portfolio composition is guided by the overarching criteria of Fit, Balance, and Impact.

1.1 Purpose

The Horizontal Fusion Portfolio consists of a wide range of member Initiatives that developed a product for the Portfolio or integrated existing products into the SOA. The Horizontal Fusion Developers Guidance Document, Developers Reference Document, and Standards and Specifications Document are designed to help new Initiatives of the Portfolio as well as existing Initiatives.

1.1.1 Developers Guidance Document

The purpose of this document is to provide guidance to current and new developers of the Horizontal Fusion Portfolio. The focus of this document is to provide brief descriptions of the most commonly used technologies including Extensible Markup Language (XML), Web Service Description Language (WSDL), and Universal Description, Discovery and Integration (UDDI) as well as the Core Enterprise Services (CES). In addition, this document will provide reference material to aid Initiative developers in their planning and development process. This document is not intended to be a 'how to' document; rather this document provides a brief overview of the technologies and guidelines on how to implement these technologies.

1.1.2 Developers Reference Document

The Developers Reference document is intended to provide comprehensive descriptions as well as technical interface documentation of the CES. The Developers Reference Document is the most technical of the three developers' guides.

The largest portion of the Reference Document is the documentation on CES. CES include:

- Security Services
  - Security Adjudication Service (Classification Policy Decision Service)
- Service Discovery Services
- Content Discovery (Federated Search)
- Mediation Messaging
- Syndication: Alerts
- Person Discovery

- Collaboration Services

Please refer to the Developers Reference document for detailed information on these services.

In addition, the Developers Reference document will provide information and reference material for any software development kits (SDK) that these providers have developed and released. Developers can leverage existing technologies and software. SDKs help developers interface with the services without having to write an extensive coding interface.

1.1.3 Standards and Specifications Document

The purpose of the Standards and Specification Document is to provide direction to the Horizontal Fusion Portfolio Initiatives towards the development of web services and data management standards. Many of these standards are geared toward developers; however, anyone involved in networking, document creation, or general file releases can benefit from this document.

1.2 Current Versions

The Horizontal Fusion Developers Guidance Document refers to the latest version of both the Developers Reference and Standards and Specifications Documents at the time of its publishing. In some cases, documents will change to incorporate updated guidelines, specifications, or technical information. Always check to find the latest approved version of these three documents.

2 WEB SERVICE TECHNOLOGIES

The following section outlines a few of the more commonly used web service technologies. These technologies and descriptions are not intended to be a full and comprehensive tutorial on Web Services; however, it is intended to help developers understand the Horizontal Fusion SOA.

2.1 eXtensible Markup Language (XML)

XML is the basic data format for web services. XML is highly extensible and used for not only data formats but also access protocols, such as Simple Object Access Protocol (SOAP). Developers should attempt to keep as much data and communication in XML format as possible. XML forms the basis for other web service technologies, such as Security Assertions Markup Language (SAML), WSDL, and the language for SOAP itself. All XML should be well-formed* and conform to the World Wide Web Consortium (W3C) Standards located at: http://www.w3c.org/XML/.

> *A document is well-formed when it is structured according to the rules defined in Section 2.1 of the XML 1.0 Recommendation, stating that elements, delimited by their start and end tags, are nested properly within one another.

When writing XML, there are several restrictions set in place for compatibility. Comments in XML may exist wherever grammar permits; however, the use of a double-hyphen, '--' is not permissible within an XML comment. This follows true for all XML documents, including XML, eXtensible HyperText Markup Language (XHTML), XML Schema Definition (XSD), WSDL, SOAP, SAML, and any other XML-based languages not mentioned. Likewise, documents must be Well Formed and adhere to the standards set forth by the W3C.

2.1.1 eXtensible Markup Language (XML) Tools

Several tools exist online to validate XML and XML-based languages. The W3C has put forth a validation service, http://www.w3.org/2001/03/webdata/xsv that will check both XML and XSD documents for validity, adherence to standards, and well-formed ness. This validation service is provided by the creators of the standard, the W3C, and should be used if at all possible.

2.1.2 eXtensible Markup Language (XML) Namespaces

XML namespaces provide a simple method for qualifying element and attribute names used in XML documents by associating them with namespaces identified by Uniform Resource Identifier (URI) references. A namespace should be unique and persistent. Each Initiative should choose a namespace before developing schema and services and maintain this consistent namespace throughout all XML documents.

In addition to the standard namespace restrictions within the standard, initiatives should not utilize the following non alpha-numeric characters: #, !, &, %, @, or spaces. In FY2004, Horizontal Fusion Initiatives found that these characters can cause problems when registering services into Service Discovery and UDDI registry. Also, uniqueness is highly important when creating an XML Namespace. The preferred format for namespaces in Horizontal Fusion is:

```
http://initiative.domain/ServiceName
```

In this example, initiative.domain is the domain of the Initiative and ServiceName is the name of the service the XML relates to; likewise both should be replaced with the appropriate values. However, for many Initiatives, this convention may not apply. For Initiatives that do not have an organizational domain, please utilize the following convention:

```
http://horizontal-fusion.dod.mil/initiative/ServiceName
```

In this example, initiative should be replaced with that particular Initiative name and ServiceName should be replaced with the appropriate service name value that the XML document refers to.

2.2 Web Services Description Language (WSDL)

WSDL is an XML-based language that describes the interface for a web service. “WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information” (http://www.w3c.org/tr/wsdl/). For interoperability purposes, each service should be contained within its own unique WSDL document. While the WSDL specification states multiple services can be located within a single document, in FY2004 Horizontal Fusion discovered that such documents would not register properly because the Service Discovery and Security Management Console used for registration is incapable of parsing multiple services within a single WSDL document. WSDL namespaces should adhere to the aforementioned restraints for XML Namespaces.

Some tools allow for WSDL generation based on source code or compiled code. Apache AXIS, for instance, is a set of Java tools that allow for the generation of WSDL from code, or the generation of some code from a WSDL. For a tool such as this, or any tool used to automatically generate a WSDL, validate the WSDL generated. Conversely, if you use a tool to build code from a WSDL, validate that the code can in fact generate a WSDL that validates and that the original WSDL used to generate this code is valid.

2.2.1 Sample Web Services Descriptive Language (WSDL)

The following snippet of code represents a very brief WSDL that has omitted schema parts. A common practice, when writing WSDL documents, is to separate schema information into a separate file that the WSDL then references. This practice allows developers to change values, data types, and schema elements without having to ensure continuity within the WSDL document. Likewise, a single schema document can then be used across many WSDL documents for several services.

All WSDL documents are XML documents and should have the same first line, including the XML version and encoding type. Other XML declaration information is optional. Some documents may work without this initial line, however, the standard for WSDL documents specifies to use of the XML declaration shown below in Figure 2.2.1 A.

```
<?xml version='1.0' encoding='utf-8' ?>
```

Figure 2.2.1 A: XML Initial Declaration

The root tag of a WSDL document is the `definitions` tag, shown in Figure 2.2.1 B. This tag contains all of the other tags and accepts parameters essential to a proper WSDL document. Namespaces should be declared within this section of the document.

```
<definitions name="sampleSearch"
  targetNamespace="http://horizontal-fusion.dod.mil/sampleSearch"
  xmlns:es="http://horizontal-
fusion.dod.mil/sampleSearch/sampleSearch.wsdl"
  xmlns:esxsd="http://horizontal-
fusion.dod.mil/sampleSearch/sampleSearchSchema.xsd"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
```

Figure 2.2.1 B: Definitions Tag

`Message` declarations, as shown below in Figure 2.2.1 C, require a name and other elements. Below is a sample message for `GetSampleRequest` and `GetSampleResponse`. The schema that represents these messages has been omitted. However, the declaration of messages occurs within the `definitions` tag.

```
  <message name="GetSampleRequest">
    <part name="body" element="esxsd:GetSampleRequest"/>
  </message>

  <message name="GetSampleResponse">
    <part name="body" element="esxsd:GetSampleResponse"/>
  </message>
```

Figure 2.2.1 C: Sample Message Declaration

`Get getSamplePortType` definitions utilize messages already declared. The `operation` tag represents the single operation this service performs, `search`. The search operation takes an input of `GetSampleRequest` and returns a `GetSampleResponse` message or a fault as shown in Figure 2.2.1 D. Services should handle null returns and fault messages appropriately.

```
<portType name="GetSamplePortType">
    <operation name="search">
      <input message="es:GetSampleRequest"/>
      <output message="es:GetSampleResponse"/>
      <fault message="es:GetSampleFault"/>
    </operation>
  </portType>
```

Figure 2.2.1 D: Sample getPort Operation

Binding references are used later in the service definition. This section helps to tie the messages, ports, input, output, and faults together into a single binding that a service can reference as shown in Figure 2.2.1 E.

```
<binding name="sampleSearchSoapBinding"
           type="es:GetSamplePortType">
    <soap:binding style="document"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="search">
      <soap:operation
        soapAction="http://www.snowboard-info.com/sampleSearch"/>
      <input>
        <soap:body use="literal" namespace="http://horizontal-

fusion.dod.mil/sampleSearch/sampleSearchSchema.xsd"/>
      </input>
      <output>
        <soap:body use="literal" namespace="http://horizontal-
   fusion.dod.mil/sampleSearch/sampleSearchSchema.xsd"/>
      </output>
      <fault>
        <soap:body use="literal" namespace="http://horizontal-

                  fusion.dod.mil/sampleSearch/sampleSearchSchema.xsd"/>
      </fault>
    </operation>
  </binding>
```

Figure 2.2.1 E: Sample Binding Code

Finally, the `service` tag defines the service as shown Figure 2.2.1 F. The `documentation` tag can be used by WSDL parsing tools to help define and describe a service. This tag is optional. The port references the previously defined `GetSamplePort` and specifies the location of this service.

```
<service name="sampleSearchService">
    <documentation>Horizontal Fusion Sample Search</documentation>
    <port name="GetSamplePort"
          binding="es:sampleSearchSoapBinding">
      <soap:address location="http://horizontal-
fusion.dod.mil/sampleSearch"/>
    </port>
  </service>
```

Figure 2.2.1 F: Sample Service Tag Definition

Finally, close the WSDL root tag `definitions with </definitions>`.

2.2.2 Web Services Descriptive Language (WSDL) Versioning

The version of a WSDL document should reflect not only the current version of that WSDL but also the version of the corresponding service to which that WSDL is written. Developers must be able to ascertain which version of a WSDL document and the corresponding service version in order to maintain compatibility. Every publicly available or distributed WSDL document will reflect this information.

2.2.2.1 Recommended Web Services Descriptive Language (WSDL) Version Numbering

The version of a WSDL (Web Services Definition Language) specification is represented as a triplet in the form #.#.#. The leading number indicates the major version, the middle number the minor version, and the final number is the annotation revision.

- **Major Number** The major number indicates a specific set of functionality. Different versions of the same WSDL with the same first number represent the same overall functionality. When major feature enhancements are added or functional change takes place, then a WSDL's major number should be incremented.
- **Minor Number** The minor number is used to represent implementation fixes and minor functional enhancements.
- **Annotation Number** The annotation number is used to reflect changes in documentation. Backward compatibility is required. This allows the WSDL documentation to be updated without impacting any implementation of that WSDL.

2.3 Universal Description, Discovery, and Integration (UDDI)

UDDI is the specification for web-based registries that expose information about a business or other entity and the technical interfaces for services this business provides. In many ways, UDDI can be considered the phone book for web services. UDDI exposes service locations, interface, and Technical Model (tModel) information. More extensive documentation on the UDDI Specification can be found on the web at: (http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uddi-spec)

A Qualified Name, or Qname, is the arbitrary key used to uniquely identify a service within UDDI. For most commercial UDDI servers, Qnames are arbitrarily chosen by the service registering person or automatically generated from the WSDL Target Namespace and other uniquely qualifying values found within the service.

Identification, classification, technical, and business information about a service is stored within a tModel. tModels are attached to a service within UDDI and reference a binding point for both the tModel instance and the service instance that institutes that tModel. Through tModels, searches can be done to dynamically discover services within UDDI that provide specific capability, adhere to specific specifications, or provide specific Quality of Service (QoS) values for a binding of a service. tModels are designed to be reusable and distributed in such a manner that service providers can utilize existing tModels and build a core infrastructure of service capabilities and interfaces common to the enterprise.

The UDDI Specification utilizes a basic taxonomy system, or classification system, to categorize services and business entities. Those registered services are discovered using manual queries or dynamically through the same UDDI Registry interface. Registry interfaces are defined in the UDDI specification and are generally exposed as web services. Manual queries can be made against an UDDI Registry to obtain a list of services matching criteria including: taxonomy search, tModel-search, and name search. Dynamic Discovery of services involves other services automatically searching the UDDI for matches to tModel-references, extracting the binding point and service information, and utilizing then utilizing that information to interface with the service. tModels are identification information used inside UDDI that are created to a particular specification. tModels identify which specification a service is implementing and usually contain a Uniform Resource Locator (URL) or URI reference to a specific implementation of that specification.

One of the major premises of an SOA is that services are both distributed and discoverable. UDDI and UDDI registries are the means by which services are discovered. As mentioned before, discovery of services can occur with manual queries or through dynamic discovery. Service providers are encouraged to utilize dynamic discovery of services to obtain endpoints for common services as well as endpoints for services providing capabilities being leveraged. Along the same lines, service providers are required to enter accurate and descriptive values into the UDDI registry so that other services can utilize the newly registered services and capabilities. The use of tModels within Horizontal Fusion has been limited; however, service providers are encouraged to develop, implement, and reuse tModels built by the DoD.

2.3.1 Dynamic Discovery

Dynamic Discovery is the process by which a service or services discover relevant services within the enterprise through the UDDI registry dynamically utilizing registration, tModel, and taxonomy information provided with the UDDI Specification.

The following example has four services. Service A provides a temperature for a given location; Service B provides the time for a given location; and Service C provides conversion between time formats and temperature formats. Each service in Figure 2.3.1 A must register with the UDDI server.

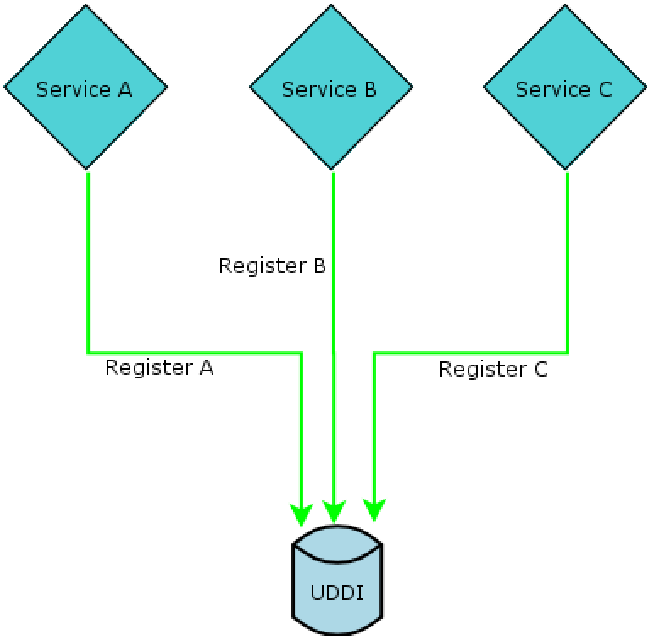


Figure 2.3.1 A: Example of Service Registration

Following registration of Services A, B, and C, a new service called Service D requires the time in Washington, DC. Instead of writing another service to determine this time, Service D queries a UDDI Registry to find other services that provide time functionality. Using the taxonomy and business models of the UDDI server, Service D queries the UDDI server for any service that will provide a time given a specific location. The UDDI server returns the list of services which-handle time services, including Service B.

Utilizing the appropriate information gathered from the tModel associated with Service B (Time Service), Service D is able to query Service B to obtain the time in Washington, DC. Figure 2.3.1 B shows the general flow of information from the UDDI search to the final data response from Service B to Service D.

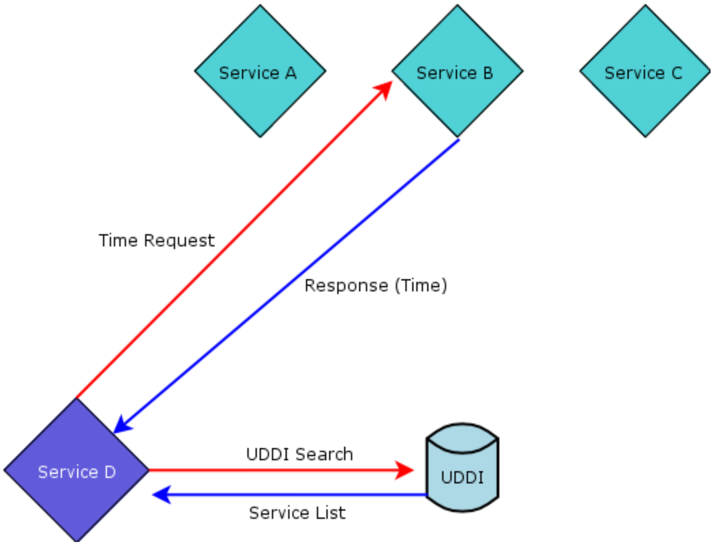


Figure 2.3.1 B: Flow of Information from UDDI Search

Unfortunately, the tModel-development to date for the DoD and Horizontal Fusion has not been implemented to such a level that Dynamic Discovery of services can be readily utilized. However, by developing a tModel-for similar track providers, there has been success with lesser forms of dynamic discovery. Initiatives are encouraged to utilize UDDI information as much as possible.

2.3.2 Universal Description, Discovery, and Integration (UDDI), Service Discovery, and Security Services

UDDI, Service Discovery, and Security Services are closely coupled in the Net-Centric Core Enterprise architecture. The registration process for services within the Horizontal Fusion Portfolio is more complex and involved than the commercial world because registration with UDDI also registers a service with Security Services. Likewise, Service Discovery is directly based on the UDDI registrations. The coupling of these three services creates a one-to-one-to-one relationship between entries stored within UDDI, Service Discovery, and Security Services. The resulting relationship creates dependencies between UDDI, Service Discovery, and Security Services and limitations upon the specifications of UDDI, WSDL, and SOAP that are not common to commercial implementations of these same standards. When at all possible, documentation about these dependencies or changes to the specifications or standards will be provided.

The NCES Service Discovery Service (NSDS) Management Console provided with Service Discovery and Security Services is used for registration in the Horizontal Fusion architecture. Figure 2.3.2 A shows the registration, binding and rule definition process for a single service. After registration, the registration administrator must then input manual binding points because the binding points located within the WSDL document are not automatically parsed. Finally, the service operations and role policy is entered to complete registration.

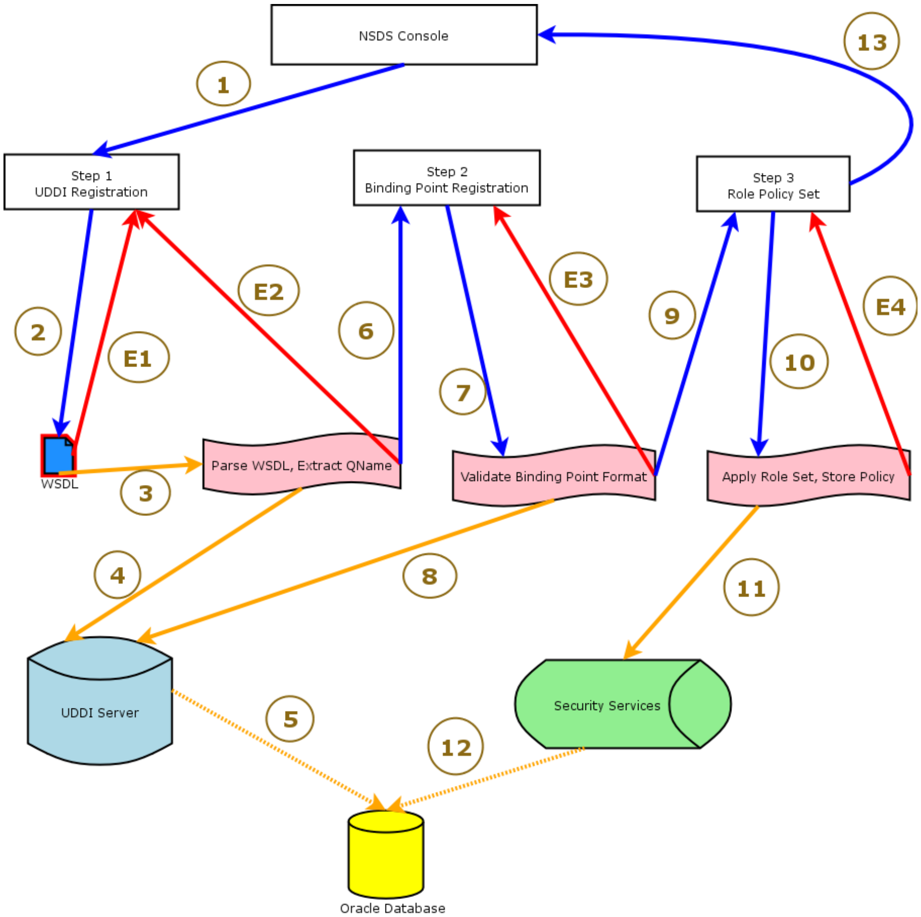


Figure 2.3.2 A: Registration Process

1) Service Registry Administrator selects the new registration link on the management console.

From this page, the Administrator will enter the Service Name, Description, and WSDL location. Additionally, the Administrator will select the appropriate categories within the UDDI Taxonomy to place the service, and chooses which of the roles from Security Services will be utilized for this service. Note, this process does not set any policy for the service it simply captures which roles will be used in the policy set in **Step 3**.

2) The console then proceeds to download the WSDL from the specified location. Should the WSDL not be obtainable, the user is returned to **Step 1**.

E2) The server was not able to obtain a copy of the WSDL from the Internet. In many cases, the WSDL is simply protected via secure socket layer (SSL) or tw0-way SSL. Registration over 1-way and 2-way SSL should be available in the near future; however, Initiatives are encouraged to have a pure HyperText Transfer Protocol (HTTP) available WSDL for each service.

3) The server parses the WSDL and obtains elements, including Qname. Qnames are generated from the Target Namespace within the WSDL Definitions and the Service Name specified in the service tag of the WSDL. The server isolates any operations declared within the WSDL for the service. Should any parsing errors occur, the user is returned to **Step 1**.

E2) The parser was unable to validate or parse the WSDL document it obtained. Note that WSDL documents must contain a unique combination of Qname and Service Name, otherwise a non-unique Qname will be generated and thus registration will fail.

4-5) The UDDI server stores the information for this service in the UDDI registry, which happens to be an Oracle Database.

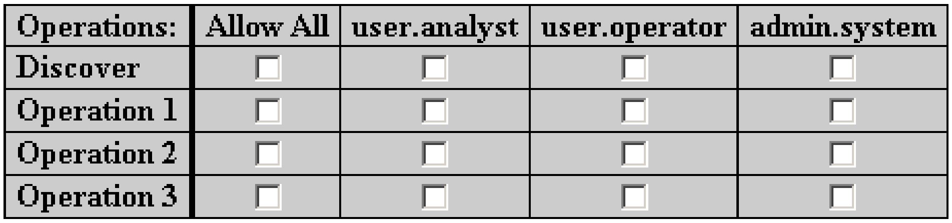
6) Since no errors have been found, the user is then redirected automatically to **Step 2**.

7) At this point, the user enters the binding point information. This information should be correct in the WSDL document, and should contain an SSL endpoint. Endpoints and Qnames should not be relative to a specific network; that is, **do not refer to localhost or local Internet protocol (IP) ranges** for a service endpoint.

8-9) Provided the user enters a valid combination of port type (http, HyperText Transfer Protocol (Secure) (https),file transfer protocol (ftp), etc) and address, the information is passed to the UDDI server and stored. The user is then redirected to **Step 3.**

E3) The Administrator entered an invalid combination of port type and address.

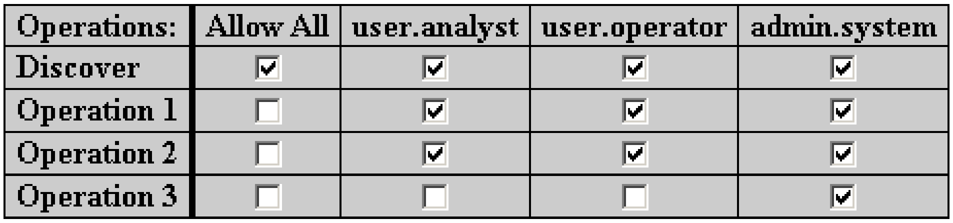
10-12) Based on the roles selected in Step 1 and operations obtained in Step 3, the server displays a matrix of roles and operations (Figure 2.3.2 B)



Operations:	Allow All	user.analyst	user.operator	admin.system
Discover	□	□	□	□
Operation 1	□	□	□	□
Operation 2	□	□	□	□
Operation 3	□	□	□	□

Figure 2.3.2 B: Matrix of Roles and Operations

The Administrator-then selects which roles are allowed. Note, Discover is included for every service and determines the role required for discovery. Administrators should allow all for discovery. Figure 2.3.2 C shows a role policy selected where any role can discover, Operation 1 and Operation 2 allow the user.analyst, user.operator, and admin.system while Operation 3 requires a role of admin.system and will not accept any other role.



Operations:	Allow All	user.analyst	user.operator	admin.system
Discover	☑	☑	☑	☑
Operation 1	☐	☑	☑	☑
Operation 2	☐	☑	☑	☑
Operation 3	☐	☐	☐	☑

Figure 2.3.2 C: Role Policy Example

13) Should the Administrator enter a proper role policy and there are no errors setting the policy, the user is redirected to the console home page. The registration is complete.

Once a service has been registered properly with UDDI, Service Discovery, and Security Services, it can be used by other services. The Policy Decision Service (PDS), a service within the Security Services, allows for easy integration of the role policies set during registration. Figure 2.3.2 D shows the role of security services in a very simple flow of service-to-service communication.

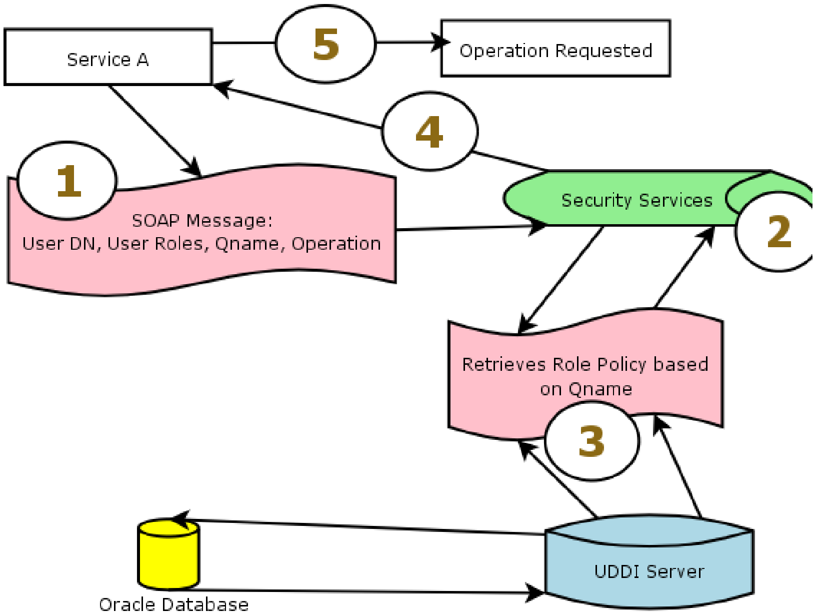


Figure 2.3.2 D: Role of Security Services

Figure 2.3.2 D demonstrates how Service A contacts PDS through Security Services padding the user’s Distinguished Name (D/N), roles, and the service Qname and

Operation attempting to be performed (1). Security Services (2) retrieves information from the UDDI server and (3) evaluates the users roles vs. the role policy established during registration. Based upon the policy, Security Services return a yes or no answer to the general question "Can this user with these roles perform this operation of the service with this Qname?" that is provided in the SOAP message. Should a positive answer be returned (4), Service A then performs the operation (5).

2.4 Lightweight Directory Access Protocol (LDAP)

LDAP is the standard directory protocol and specification used to store user information for authentication and identification purposes. The Person Discovery services are integrated with the Defense Information Services Agency (DISA) Global Directory Service (GDS). GDS contains identification data about all users who have received DoD PKI certificates. GDS is implemented in a Netscape LDAP directory. The core LDAP has the schema defined by the GDS. This core schema did not support all the requirements for Discretionary Access Control-Plus (DAC+) or the Person Discovery capability. Horizontal Fusion, in coordination with the DISA GDS program, extended the schema to include clearances, roles, phone numbers, expertise descriptions, email addresses, and areas of expertise.

LDAP can be accessed via an unsecure LDAP protocol or the secure LDAPS protocol. Any services that connect to LDAP directly must do so using the LDAPS interface, similar to the difference between HTTP and HTTPS. Also, Horizontal Fusion security precautions prevent an '*anonymous bind*' connection to LDAP, thus all connections to LDAP require the proper read-only user provided to the enterprise.

3

CORE ENTERPRISE SERVICES (CES)

CES are capabilities that create the foundation and infrastructure of the SOA. The functionality of these services is designed to be leveraged. In order to be considered SOA compliant in the Horizontal Fusion Architecture, service providers should leverage the Core Enterprise.

The overview information provided in this document is not meant as a technical guide. Initiatives should refer to the Developers Reference Document for more information on CES as well as technical information on these services.

3.1 Security Services

The Security Services will perform authorization and authentication of all systems within the execution chain. A common enterprise security model ensures greater service interoperability and relieves each application from having to implement a separate security model. The CES Security model is based on a number of open standards, (i.e. Organization for the Advancement of Structured Information Standards (OASIS), W3C, SAML, eXtensible Access Control Markup Language (XACML)) and is implemented with a PKI solution.

Security Services are fully exposed as web services and the primary functions include authentication, authorization, and ability to off load PKI related functionality. Authorization is based on a role policy and requires policies to be associated with various resources and or principals. These policies will specify access rules based on the particular role of a user.

The basic Security Services will need to be leveraged by both the service provider and the service consumer. A set of security specifications detail a roadmap on which services are to be invoked and how they are to be invoked to ensure security. A service consumer and service provider platform could actually write code to directly invoke the necessary services outlined in the specifications. However, for simple entry into the enterprise security system, an SDK was developed which implements the necessary security calls for both the service consumer and provider. This SDK can be installed and deployed on specific platforms. When the SDK is installed and deployed it will automatically make the necessary invocations to ensure security. Therefore, the service provider and service consumer applications will not need to explicitly call Security Services. However, the SDK for these services is not required to interact with the services.

Security Services, especially the Policy Decision Service (PDS) should be utilized by Initiatives. Initiatives should refrain from hard-coding or self-coding policy decision.

3.1.1 Security Adjudication Service (Classification Policy Decision Service)

The Security Adjudication Services are services that extend the Security Services. Classification Policy Decision Service (cPDS) is designed to handle classification of data and user access to data and services based upon clearance and citizenship information. cPDS includes several parts, including a handler, SDK, and the Security Adjudication Service. The handler and SDK both utilize the Security Adjudication Service. Among the services provided by cPDS are the ability to:

- Validate an input security level
- Determine the relationship between two input security levels, a Subject Clearance and an Object Classification, which can be one of the following: Equal, Subject Strictly Dominates, Object Strictly Dominates, or Incomparable
- Determine the aggregate of a list of security classifications to determine the clearance level required to read the objects whose classifications were used as arguments
- Determine a group clearance level from a list of user clearances
- Determine if data can be released based on the data's classification and a list of user clearances
- Determine if data can be received based on the user's clearance and a list of data classifications

More information on how to use the Security Adjudication Service can be found in the Horizontal Fusion Developers Reference.

3.2 Service Discovery

The Service Discovery Service provides access to an enterprise level directory of existing services. Through this service, consumers can register new services and add bindings to these services as well as perform dynamic lookups and searches of the available services already available. Service discovery provides access to a service directory built on the UDDI 2.0 specification. The Service Discovery Services are grouped into two categories—Service Publishing and Service Inquiry.

More information on how to use the Service Discovery Service can be found in the Horizontal Fusion Developers Reference.

3.3 Content Discovery (Federated Search)

The implementation of the Content Discovery capability in Horizontal Fusion is not a single application. It is the creation of a virtual information space, called Collateral

Space. Collateral Space is defined by data providers and routing engines that will guide queries and responses along the most contextually significant paths.

Access to the Collateral Space, from the query generation perspective, is obtained by the utilization of the Search Web Service (SWS). It is worth noting that not only end users may generate queries, but also any point in the Collateral Space can and will use the SWS to send queries and obtain results. Alternatively, data providers utilize the Registration Web Service (RWS) to become part of the Collateral Space.

If the SWS and RWS are the glue that connects data providers and query generators, the Intelligent Federated Index Search (IFIS) provides the map that directs queries through the Collateral Space. In simplistic terms, the workflow of IFIS may be viewed in the following steps:

- Receives a query—natural language or keyword or Community of Interest (COI) specific
- Normalizes that query into a standard form by removal of natural language ambiguity based on a war-fighter's roles and personal preferences
- Recognizes semantics of common terms and keywords
- Determines the proper routing to access the relevant data providers (based upon information contained in the registry of data providers)
- Sends the normalized queries to the pertinent data providers via the SWS
- Receives the results from the data providers, again via the SWS
- Performs a variety of post-processing steps to order and enhance the results for presentation to the query generator
- Returns the results

On top of the IFIS engine is a presentation layer that is essentially de-coupled from the process described above. In fact, IFIS may not return the results to a user interface at all, but simply to another instance of IFIS or other automated system.

Content Discovery, as implemented by the Federated Search, is a powerful next-generation knowledge discovery framework that permits authorized users and organizations on the collateral network the ability to search a vast array of indexed, non-indexed, structured, and unstructured data using a single point of entry. This isolates the end user from the task of querying multiple sources and from having to correlate information after the fact. All data is exposed in a uniform manner and only those data providers that are relevant to a given query are searched. In addition, Federated Search supports role-based security, allowing or blocking data based upon the authorization credentials of the user. The user may also customize query normalization and query routing by defining a search profile. This allows for User biases and a prior knowledge to be incorporated in the search. IFIS implements a network-centric information querying system that understands the military's use of short hand expressions, representations, and acronyms. This language usage understanding allows IFIS to semantically enhance a query according to its context and smartly route the enhanced query to its data sources depending on their coverage and product type. It takes advantage of the large set of data already available in existing DoD data stores.

A lot of information is often obscured by the simple fact that different organizations call the same thing by different names. In addition, acronyms can further obfuscate the query. IFIS employs a large set of semantic ontology's to accurately cast terms into a common lexicon.

More information on how to use the Content Discovery Service can be found in the Horizontal Fusion Developers Reference.

3.3.1 Core Taxonomy

The Horizontal Fusion data providers collaborated on a discovery taxonomy. This taxonomy was provided to the ASD/NII Taxonomy Focus Group (TFG) and became the basis for the group's core taxonomy. The TFG mapped the Horizontal Fusion taxonomy into core concepts and had functional experts augment and de-conflict concepts as needed. In addition, rigorous taxonomic conventions were applied.

The core taxonomy serves as a hub for the DoD taxonomy framework. COI taxonomies are currently being plugged into this framework to provide an extensible DoD taxonomy. The resulting ASD/NII core taxonomy was used in both data source registration for Federated Search, as well as query result visualization.

The taxonomy can be found on the Horizontal Fusion workspace: https://workspace.asnrdacheng.navy.mil/DataMgtWG/Ontology%20Focus%20Group%20Library/Forms/AllItems.htm

3.4 Mediation/Messaging

The Mediation/Messaging Service provides a federated, distributed, and fault-tolerant enterprise message bus which:

- Delivers high performance, scalable, and interoperable asynchronous event notification including alerts, track updates, etc., to both applications and end-users using multiple messaging models including publish and subscribe, queuing, and peer-to-peer
- Provides QoS including priority, precedence, and time-to-live
- Provides guaranteed delivery to disconnected users or applications

- Utilizes multiple message brokers, potentially within different administrative domains, to support the distributed, federated nature of the GIG

A number of standards have emerged which allow for the synthesis of a robust messaging system using web services. The foundations of the messaging specification rest with the Web Service Reliable Messaging (WS-ReliableMessaging) and Web Service Addressing (WS-Addressing) standards that allow the expression of required QoS parameters for guaranteed delivery and decouple the destination of a message from the underlying transport, respectively. Both the WS-ReliableMessaging and WS-Addressing standards require support from the client web-service libraries. The WS-ReliableMessaging requires the client to actively engage with the server, whereas the WS-Addressing standard simply requires that the client set elements in the SOAP header. Client web-service libraries may automatically place the appropriate WS-Addressing elements in the header (for example: <wsa:Action/> and <wsa:MessageID/>). In lieu of client library support, the client application can simply add the header element where appropriate.

More information on Mediation Messaging is included in the Developers Reference Document.

3.5 Syndication: Alerts

Horizontal Fusion offers an Alert and Subscription capability that is designed to improve information sharing and situational awareness. Portal users can easily create new topical subject channels to share data on fast breaking events and can manually generate Alert/Notifications with new information or analysis to any topical channel ensuring all interested parties of that topic receive the information. Portal Users subscribe to "Topics" to receive the alerted information. Alerts/Notifications are sent to the user's Alert portlet with a short description and web link to the content.

In addition to receiving alerts from the Portal, any approved external system can send alerts to the Syndication Server. This document describes the Web Services that the Syndication Server offers, and how to connect with, send and receive Topic/Alert information.

With the advent of the Messaging Application Program Interface (API), the Syndication Server now works to receive and broadcast alerts across the Messaging solution.

Because the Syndication and Messaging API Servers are based on Web Services, this document is not intended to be a tutorial on Web Services. It is aimed at experienced developers.

More information on how to use the Syndication:Alerts Service can be found in the Horizontal Fusion Developers Reference.

3.6 Person Discovery

The Person Discovery Services provides support for a complete and fully integrated collaboration capability. These services are now integrated with the DISA GDS, and the Horizontal Fusion Federated Search. The Horizontal Fusion LDAP directory has been extended to allow for integration with GDS as well as to contain all of the following information expert attributes, roles, groups, and clearance information. This LDAP directory is initially loaded with data exported from the GDS directory and subsequently updated via an automatic process as changes occur in GDS.

The Person Discovery Services provide a layer of abstraction to all person discovery service consumers from LDAP connectivity issues LDAP schema changes. Furthermore, Person Discovery Services leverage WS-Security to allow users access to only the discovery services that they are entitled to use.

More information on how to use the Person Discovery Services can be found in the Horizontal Fusion Developers Reference.

4 SOFTWARE “BEST PRACTICES

This portion of the Developers Guidance covers some of the Software “Best Practices”, or commonly used practices that aid in the production of higher quality software. While there exist many “best practices”, this document will cover Source Code Control Systems, defect tracking, testing, and documentation. Also, this section will identify some of the development techniques and processes. Likewise, this section will identify some of the usability faults and provide tips to alleviate these faults.

It is important to remember that practices alone do not establish a process. Specifically, software development practices encompass “what” should be done whereas a software development process establishes “how” to do it. Simply put, a process is the implementation of a set of practices. Practices are generic and common across a community; however, processes are typically organization-specific.

The intent of the following sections is to provide a simplified introduction to some of the general practices in software development. With this as a starting point, understanding the volumes of texts that exist for each practice will be a manageable undertaking. Additionally, understanding these general practices will leave one more informed when researching or considering process improvement tools.

4.1 Source Code Management

Source code is the most fundamental component of a software product; it is the recipe for software construction and functionality. Managing the execution of a software project fundamentally begins with managing revisions to the source code. Poor source code management is evidenced by software that is easily “broken” by a recent change, but cannot be easily “rolled back” to the previously working version. Some “buggy” software achieves that state when potential or actual defects are identified, but not tracked. The following sections present some mitigating practices for these symptoms and their underlying cause—poor source code management.

4.1.1 Source Code Control Systems

Source Code Control Systems (SCCS) are used for managing repositories of source code and the revisions to that source code. At its most basic use, an SCCS tool provides a facility for source code storage and maintains a historical record of the source code. The fundamental rule of SCCS tools is that nothing is ever destroyed, which is why SCCS tools are useful; every version and iteration is efficiently stored in a readily accessible manner. Additionally, many tools allow the user to submit comments with each change, which provides revision history for the code. By using some of the more advanced capabilities in some SCCS products, developers can separate and distribute the effort, leading to increased efficiency from parallel execution. A typical SCCS tool consists of two separate components 1) a server that controls the central storage of the code and 2) a client that interacts with the server to obtain working copies. Often the term CVS, which stands for Concurrent Versioning System, is used as a synonym for SCCS; however, CVS is a specific type of SCCS and thus all SCCS should not be referred to as CVS.

An SCCS system relies on two areas of storage 1) the repository and 2) the working folder. The repository is the master storage location for the source code and is completely managed by the SCCS server. It archives all versions of software submitted via the SCCS client. It maintains the latest version of the code and all of the necessary deltas to revert that code to previous versions within the history. The server component of the SCCS tool provides the interface through which all interactions with the repository must flow. The working folder is a local copy of the contents of the repository used for working on the source code. The client component of the SCCS tool interacts with the server to manage the contents of the working folder based on user requests.

Two common methodologies exist for the interaction between the server and client components 1) Check Out, Modify, Check In and 2) Edit, Merge, Commit. In some cases, the SCCS tool dictates the interaction method; in others it is a user configurable option.

4.1.1.1 SCCS Process: Check Out, Modify, Check In Workflow

Tools using a check out, modify, check in workflow typically use four main operations Add, Get, Check Out, and Check In. The Add operation adds new files to the repository for future tracking purposes. The Get operation, sometimes referred to as "Update," obtains a copy of the current version of the file and places that copy in the working folder. This operation replaces the copy currently in the working folder without indicating intent to modify the file to the server. The Check Out operation indicates to the server the intent to work on a particular file. This operation usually implies a Get operation to update the working folder's copy to the latest version of the file. In some tools, this causes the server to prevent other users from performing check-outs. Finally, the Check In operation indicates to the server that updates to the file are complete and the client sends the updates back to the server to modify the content of the repository.

The workflow typically consists of performing a check-out on the file(s) being edited, making the necessary edits to those files, and then checking them back into the repository. For efficiency, a Check Out, Modify, Check In model typically imposes a few rules on the process and related tools:

- The working folder cannot be edited directly and contains read-only content
- Developers must check out files to make modifications
- Check-outs must be made with exclusive locks so that only one developer can modify the contents of a particular file at a time

For example, during a Check Out operation, the client indicates that the user is requesting a file from the repository. The SCCS server checks the current status of that file and if it is not locked, sends the update to the client. Meanwhile, the server denotes the file's current use status and allows the user to lock the file for exclusive use. The client software receives the update, replaces the local working folder version with the update, and marks the file as editable. The user proceeds with updates to the file or abandons changes. If the changes are abandoned, the working copy is reverted back to the version in the repository, deleted, or left as-is, depending on user preference.

When modifications are complete, the user performs a Check In. During this operation, the client sends the updated file to the server and restores the read-only state of the copy in the working folder. The server receives the update; modifies the content of the repository; increments the version number; and removes the checked-out state placed on the file, including any exclusive locks on that file.

Because of the aforementioned rules, the Check Out, Modify, Check In workflow is considered a traditional and conservative approach. Proponents of this method believe that multiple developers modifying a file concurrently cause chaos and results in additional errors or lost productivity. However, it creates a situation where developers could be stuck waiting for a locked file before being able to continue work. Therefore, the following tips should be considered when using a check-out-based process:

- Developers should only check-out those files specifically needing modification
- Developers should not hold exclusive locks on files for longer than needed
- Files should be checked back in when the modifications are complete
- Developers should not leave outstanding locks on files unless they are actively using them (e.g., Do not go on vacation with locked files)

4.1.1.2 Edit, Merge, Commit Workflow

The Edit, Merge, Commit workflow is considered a more liberal SCCS model, avoiding most of the rigid structure of the check-out workflow. The Edit, Merge, Commit workflow offers valuable advantages, such as the ability to make concurrent changes to the same file and work in "offline" mode, which does not require an active connection to the server. This workflow is made possible by tools that can automatically merge two, or more, versions of a particular document automatically. Developers who choose the check-out workflow usually do so because they are not comfortable relying on the merge capabilities of the Edit, Merge, Commit workflow. Despite the potential for risk, the merge tools have proven themselves effective over time. The edit workflow is now the most common workflow used.

A developer using this workflow makes changes to the working file, merges edits with the latest version from the server, and commits the updates to the repository. This methodology means that all files in the working folder are always editable, which allows "offline" work. However, it also means that developers do not know who is currently making modifications to a file and therefore they must merge their updates with the current version on the server to make sure their changes are based on the most current version. In practice, however, it is rare that two users are modifying the same file, in the same location, at the same time. Nevertheless, it makes sense to stick to a singular task when making modifications to files and avoid bundling too many changes into a single update. The advantages are that other users will have the most up-to-date versions as soon as they are ready, there will be extensive documentation of what has changed because of comments inserted for each update, and if there is a problem with an update, it becomes possible to revert back in small increments and isolate the cause of the newly discovered problem.

4.1.1.3 Advanced Source Code Control

Software Configuration Management tools are not limited as a repository for code; the advanced features of SCCS tools facilitate software build and release management, versioning, and structured development processes. For example, some SCCS tools support managing "atomic transactions," which group a set of updates to the server together as one update instead of several individual file updates. This allows the developers to modify the repository structure as needed, for example, changing the folder structure of the archive is treated as a single update instead of numerous individual deletions and additions. This saves time because using the repository does not require formal meetings prior to use to determine the optimal content structure, nor does it require the wasted storage from attempting to simulate a folder move.

As mentioned previously, SCCS tools allow the developer to submit a comment describing each update in detail as it is submitted. This comment system operates on a per file basis and does not maintain logical grouping of files for versioning for the whole software package. However, through labeling, one of the advanced tools available in SCCS packages, developers can do just that. Adding a label, sometimes known as a tag, to a set of files effectively bundles them together as a single unit, taking a "snapshot" of the current contents of the repository. For example, a set of files can be labeled "Alpha 1," "5.2.03," "Dec. 19th Build," or some other meaningful label for that instant. Using these labels, it is possible to revert to a versioned set of files at any time in the future.

Branch and Merge operations allow for efficient build control by allowing flexibility in selecting specific baselines of code on which to work. The Branch operation creates another reference to the set of files in the repository and tracks those changes independently of the changes to the "main" copy of the source code. In essence, this is akin to making an additional copy for reference or experimentation. The Branch operation saves storage space in the repository over a simple copy by creating independently tracked references to files instead of duplicating them. These references provide the flexibility of working on an "alternate" version that is not necessarily part of the main development path. Two examples of using branching are release management and exploratory development. As a release management example, Version 1.0 of the software is ready for release, so the development team creates a branch for the Version 1.0 Release. This allows the team to begin developing Version 2.0 while preserving the code base for the Version 1.0 Release. The Merge operation allows changes and updates made in the branched version of the code to be merged back into the main "trunk" baseline of the source code. An example of software version branching is shown in Figure 4.1.1.3.

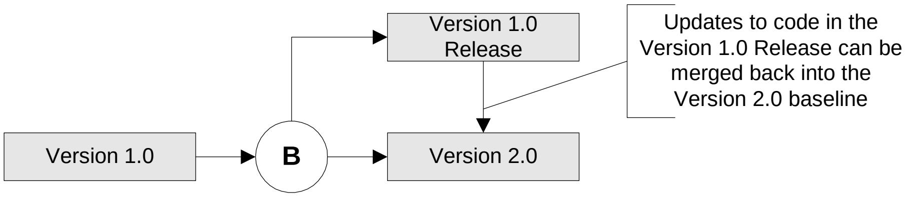

Version 1.0 Release
Updates to code in the Version 1.0 Release can be merged back into the Version 2.0 baseline
Version 1.0
B
Version 2.0

Figure 4.1.1.3: Software Version Branching

By using the Branch operation to perform exploratory coding, theories for changing the code or overall design can be tested independently of the current version of the code. If the changes are successful and beneficial, then the updates can be merged into the main track of development (the "trunk"), if the changes are not desired, the branch can be abandoned, saving the work of trying to revert to the older version. It also has the benefit of not affecting the main baseline while making experimental changes, thus reducing the potential risk associated with pursuing the experimental path.

4.1.1.4 Sample Software Configuration Management Tools

Although statistics indicate as many as 70% of software development projects do not use an SCCS tool, the practice is gaining adoption. The growing use of SCCS tools means more tools become available. Table 4.1.1.4 is not a comprehensive list of SCCS tools, but represents some of the better known tools.

Application	Product URL
Bitkeeper	http://www.bitkeeper.com
ClearCase	http://www-306.ibm.com/software/rational/offerings/scm.html
CVS	http://www.cvshome.org
Perforce	http://www.perforce.com
Subversion	http://subversion.tigris.org
Vault	http://www.sourcegear.com/vault/index.asp
Visual SourceSafe	http://msdn.microsoft.com/ssafe/

Table 4.1.1.4: Sample SCCS Tools

Bitkeeper—Supported on Windows, Linux, Mac OS/X, and Unix, provides a distributed architecture for SCCS. It is currently used to manage the source code for the Linux kernel. It is free for open source projects, but charges a per developer license for commercial projects.

ClearCase—Typically used for large organizations' projects, offers features above and beyond source code control to include build management, software asset management, and more.

CVS—Concurrent Versioning System (CVS) is used extensively and is possibly the most widely used of the SCCS servers. It follows the client-server model as previously described and is specifically designed for the edit, merge, and commit workflow. It is not as feature-rich as some of the other tools; however, many third-party tools and clients are available for it. This was the SCCS used for code submission to the Test and Integration lab for FY2004.

Perforce—Perforce is also a client-server model, but it is specifically designed for speed. Despite being engineered for speed, it does not lack capabilities; it allows some advanced SCCS features, such as atomic transactions and complex branching structures. It supports a wide array of client Operating Systems, so it is very well suited for cross-platform development.

Subversion—Subversion has been designed to be a "better CVS." It supports many of the advanced features that CVS does not, including atomic transactions, efficient handling of binary files, and arbitrary file properties.

Vault—Vault is another client-server model SCCS server, but it supports both the check out, modify, check in workflow and the edit, merge, and commit workflow. It utilizes MS SQL Server as a database backend and communicates via web services. In addition, it supports finely granular security settings and integration with the FogBUGZ defect-tracking tool.

Visual SourceSafe—Visual SourceSafe is the SCCS tool that Microsoft ships to customers with its development platform. It is noted as being inefficient over the Internet and for its lack of use within Microsoft for their projects.

4.1.2 Defect Tracking

4.1.2.1 Overview

Inevitably, over the course of software development, defects will be discovered. Conversationally, defects are referred to as "bugs." "Bugs" avoids the negative connotation of referring to the software as potentially "defective". "Bugs" has become a universal term describing defects of sorts within a development project and may include problems with source code, functionality, usability, visualization, spelling, or user-interface. Some defects are difficult to isolate, especially if they deal with a highly technical aspect of the application.

Defects can be discovered during any phase of the process—from developer unit testing to end-user feedback. Typically, these defects are reported directly to the developer via e-mail, phone, instant messaging, etc. However, these traditional methods of relating issues to a developer reduce defect visibility. Defect tracking tools consolidate all of the defect reports into a single location. The benefit of this is twofold. First, the defect-tracking tool ensures that defects actually get resolved and are not simply forgotten. Second, it provides the developers a snapshot of the common errors or mistakes made during development, which becomes valuable for any development improvement process.

The core unit of any defect-tracking tool is the defect report. A typical report contains information about the defect including, but not limited to, an assigned reference number, current status, historical status, and priority. Table 4.1.2.1 A lists some of the common report fields.

Field	Description
ID/Reference Number	Unique Number for referring to a specific defect report
Project	Name of the application/project affected by this particular defect—the project is any program or program version that is separately tracked within the tracking tool
Area	Specific component or feature of application that contains the defect
Title	Descriptive name given to defect, typically describes the problem observed or the name of the change requested
Assigned	Indicates to whom the current defect report is assigned for resolution
Status	Indicates current resolution status of the defect, generally this is open or closed with a specific resolution state (e.g. OPEN (New) or CLOSED (Resolved – Fixed))
Priority	The assigned priority for the defect's resolution efforts—it is an indicator of the defect's criticality within the project
Version	The version of the software in which the defect was discovered, this may be just the build number, or a full version number (e.g. 5.2.23 or Build 1219)
Computer / Platform	Identifies the specific computer on which this defect was discovered, especially helpful in cases where a defect can only be reproduced on specific machines (usually a client-configuration issue)
Description	The most extensive part of the defect report, contains the initial information used in opening the report and the history of resolution states

Figure 4.1.2.1 A: Common Defect Report Fields

Different defect tracking systems offer different fields, but the previous table includes some of the most common ones. It should be noted much of the content and functionality of any specific defect report is tied to the particular defect tracking tool. Despite tool-specific content, some common terms are used to indicate resolution status:

By Design/Not a defect—the program works exactly as the developers intended; however, the user or tester reported a defect. This type of defect report, while not a defect, should be considered a flag for issues regarding documentation or usability. For example, a form validation field may automatically clip values to the maximum value allowed—replacing the current value of the field. Since this was not expected, the user reported it as a defect. In this case, the documentation could be updated as a resolution to the defect or a "Max – 999" added inline beside the field. In some cases, a "By design" defect may be indicative of a poor design choice and should be considered a feature request and not merely a closed issue.

Postponed—the defect exists in a portion of the code that is being overhauled or updated in a planned release and the priority is sufficiently low enough to wait until those code modifications are made to correct this problem

Not Reproducible—the defect cannot be reproduced with the information given. This status should not be used as an easy way out, but merely a way to get more clarifying information from the tester. Additionally, the developer should start thinking about what could potentially cause the error, even if it cannot be readily reproduced.

Reassigned—the defect has been reassigned to another person for resolution. Care should be taken when using this status, as developers will sometimes bat a problem back and forth with no one taking responsibility for resolving the defect. A defect tracking system is only effective if everyone shares in the goal of shipping a high quality software product.

Won't Fix—the defect will never be fixed. Typically this occurs when the risk posed by a defect is much more minor than the cost of trying to fix it or the defect exists in a feature that will be removed or replaced in a future version. Use of this resolution should be extremely rare. It is more likely that a defect will be postponed than assigned to the "won't fix" group.

Fixed—the most common resolution for a defect, namely that the cause of the problem has been located and fixed. If the symptoms of the defect were fixed instead of the cause, the defect should be postponed and noted as having been patched to fix the symptoms covered by this defect report.

The example defect report below contains some of the elements indicated above and the flow of the defect tracking process:

1. A tester finds and opens a defect or takes a user reported defect and enters it into the system
2. A manager establishes the priority of the defect fix relative to the others in the system
3. The development team resolves the defect

4. Resolved defects are returned to a tester (typically the one who opened the defect report)
5. The tester either closes the defect or reactivates it – returning the resolution activity to the development team as shown in Figure 4.1.2.1 B.

ID	1219
Project	Chief Engineer Coffee Server 2.0
Area	Flavor Engine
Title	Coffee tastes like gas-station coffee
Assigned	CLOSED
Status	CLOSED (RESOLVED – FIXED)
Priority	2 – Must Fix
Version	2.0 Build 0891
Computer	JTF Workstation 12; Windows XP, 512MB RAM
Description	**08/01/2004 Opened by Jill the Tester** Chief Engineer reports coffee tastes like it was picked up at a cheap gas station Reproduction steps: • Lift coffee cup from table • Raise cup to lips • Take sip of coffee • Lower coffee cup to table Bug: Observe lack of flavor in coffee, specifically a distinguished lack of sweetness Expectation: The coffee should have a sweet flavor **08/01/2004 Assigned to Rusty the Lead Developer by Jill the Tester** I think this is somewhere in Joe's code, it seems to be somewhere in the getFromStarbucks( ) call. **08/01/2004 Assigned to Joe the Programmer by Rusty the Lead Developer** Joe, can you make sure that the getFromStarbucks() call is obtaining the right values internally? **08/02/2004 RESOLVED – FIXED by Joe the Programmer** Okay, I found the problem – it was an off by one error: the Barista engine was only adding two Splenda packets instead of three. **08/02/2004 Closed by Jill the Tester** Correction confirmed – the Chief Engineer reports that the level of sweetness is correct.

Table 4.1.2.1 B: Sample Defect Report

Every good defect report should, at a very minimum, include the steps necessary to reproduce the problem, the expected behavior of the system, and what actually happened. The reproduction steps are critical to the developers because those steps are their only means to track down and isolate the cause of the problem. The expected behavior needs to be documented—if there is no expected behavior, then the developer does not know if the observed behavior is actually a defect or not. Finally, careful detail of what happened is needed because that is the information about the defect that the developers require to track it. If nothing happened, then there is no defect.

4.1.2.2 Defect Tracking Best Practices

Each defect report should contain one and only one defect. Since the system is designed to track defects individually, grouping bugs together in one report defeats the advantages of using such a tool. Advanced tools allow defects with the same root cause to be grouped together.

Every defect needs to be assigned to exactly one person at a time, until it is closed. This resolves any ambiguity about who is responsible for resolution of the defect.

Resolved defects should go back to the person who opened the defect report for closure. Specifically, a defect should not go away just because a programmer thinks it should.

A good tester strips down the reproducibility steps to the absolute minimum necessary to reproduce the defect.

If the defect tracking tool used supports integration with an SCCS tool, be sure to use that integration—this allows code submissions to be tracked in response to defect reports. To manage the defects appropriately, developers should only work on defects that are in the defect tracking tool. This keeps all defects organized in one place. If a developer finds a defect during development, it should be entered into the defect tracker database to manage it.

If the defect tracking tool supports the addition of fields, avoid the temptation to do so. Although "just one little field," may not be a burden to those entering defect reports, it sets a precedent and eventually the number of fields makes filing a defect report too cumbersome. At that point people will start bypassing the defect tracking system simply because it has too much overhead to manage the defects. This frequently occurs when the defect tracking system is used *as* the software management process instead of a *component* of the process.

4.1.2.3 Sample Defect Tracking Tools

The list of available defect tracking tools varies on a daily basis as many groups develop their own tool to support their specific needs. A list of many available tools is located at http://testingfaqs.org/t-track.html. A few web-based tools are listed in Table 4.1.2.3.

Tool	Product URL
Bug/Defect Tracking Expert (BTE)	http://www.bugtracking.com/
Bugzilla	http://www.bugzilla.org/
FogBUGZ	http://www.fogcreek.com/FogBUGZ/
Mantis	http://www.mantisbt.org/
Scarab	http://scarab.tigris.org/
JIRA	http://www.atlassian.com/

Table 4.1.2.3: Sample Web-based Defect Tracking Tools

Bug/Defect Tracking Expert—BTE is an extremely flexible defect tracking tool which can be run on multiple platforms using any Open Database Connectivity (ODBC) compliant database as a backend, even MS Access. The software, in addition to being platform independent, can also be customized with scripting languages, XML, Java, and ActiveX components to extend the capabilities of the tool. In addition to many defect tracking capabilities, it offers integration with reporting and e-mail systems as well as a well-rounded defect tracking solution. BTE boasts a large client base, including but not limited to AT&T, Intel, Citibank, Underwriter's Laboratories, Nortel Networks, Compaq, and Bank of America.

Bugzilla—Bugzilla is commonly used within the open-source community since it has been tested under heavy fire as the defect tracker for the Mozilla project. It offers some basic reporting capabilities, feature request tracking, time tracking, and e-mail integration. It requires a Perl interpreter and MySQL database server.

FogBUGZ—FogBUGZ is noted as being very friendly and easy-to-use in addition to offering some powerful integration with many of the common SCCS tools including CVS, Subversion, Perforce, and Visual SourceSafe. It also has the benefit of being friendly on the wallet without necessarily being "free". It handles entries as cases and in doing so provides the ability to manage feature requests, defects, and to some extent, customer relationship management. In addition to tracking requests and defects, FogBUGZ contains the concept of an "inquiry," which, while not a defect, captures questions about why something is done a certain way and possibly sheds light on future beneficial features. It also allows importing bugs from Bugzilla.

Mantis—Mantis is a platform independent, HyperText Preprocessor (PHP)-based defect tracker. It is licensed under the General Public License (GPL) and is thus free to use, distribute, and modify to suit organizational needs. It has the ability to integrate with e-mail, perform basic searches, and manage multiple projects. It requires a PHP interpreter and a MySQL database server.

Scarab—Scarab poses a similar feature-set to Mantis, but is developed primarily using Java Servlet technology. It is also a free tool that supports XML imports and exports to and from Bugzilla. It differs from Mantis in its support for multiple database engines.

JIRA—is a Java 2 Platform, Enterprise Edition (J2EE)-based issue tracking and project management application developed to make this process easier for your team. JIRA has been designed with a focus on task achievement, is instantly usable, and is flexible.

Because there are so many tools available for defect tracking, choosing one can be a difficult process. To weed out some of the possibilities, consider the following issues:

- Consider the tool cost, both in terms of purchase price/licensing, as well as set-up and management. How do the costs scale with the development team size?
- Is multi-platform bug tracking an issue? Is your team in geographically disparate locations? In these cases, a web-based tracker may be the most beneficial solution. Even if it is not an issue, the web-based trackers provide additional flexibility in geography and potentially allow external submission of feature requests and issues.
- Can the tool be customized to include the information based on organizational needs? If not, choose a tool which contains all the foreseeable required features.
- Where does the tool store information? Many tools utilize a database backend that may not be compatible with systems currently in use (e.g. selecting an SQL Server-based tool when the organization runs on Oracle servers)
- Does the tool generate automatic e-mail notifications to focus users' attention to current issues?
- Can the tool distinguish between defects and feature requests? If not, tracking them independently would require two repositories of "to-dos".
- Does the tool offer any integration capability with the SCCS tools? This will allow defects to point to the exact submissions where the correction was made.

4.2 Testing

Testing code before final submission is essential to proper development. Each phase of development should include vigorous testing to validate not only successful cases but also any gaps in logic that can exist from failures. Development testing is different from functional testing, as the data result set is less important than the functionality of the code.

Errors that are found should be reported and fixed through the proper content management process. Isolate the error by repeating the process that generates the error.

A few good tests to run on any code based or web based system are:

- **Test the limits and constraints of user-selectable values**. In each case, ensure the incorrect values or constraints generate a proper error message and not crash the system.
  - ❑ Enter non-alphanumeric characters into a text box. While the user can be instructed to not do this, ensure that the user cannot crash the code and/or system by entering 'garbage' into text fields
  - ❑ Enter invalid dates, ranges, and other values to test for values
  - ❑ Enter code into text fields. Enter SQL, JavaScript, and HTML into fields
- **Have someone who is non-technical use the interface.** At some point, a developer will in fact do things correctly because they know how the code is expected to behave. Use someone who does not have this insight to test for errors in the code.
- **Repeat tests.** Repeat tests and change the data entered for each test. Getting a single set to work repeatedly does not cover the many code-paths that need to be tested.

4.3 Documentation

Documentation includes not only published information for end-users, such as manuals and application help, but also includes items such as inline directions, API specifications, developer references, and source code comments. These artifacts allow development teams to communicate information to the user in a beneficial format and facilitate sharing of design and intent information within and among teams. When documentation is used effectively internally, it increases efficiency because everyone involved clearly understands project intent and current progress. Externally, quality documentation improves application usability and eases user frustration when assistance is required.

4.3.1 End-User Documentation

End-user documentation covers any information provided to aid the user's understanding of the application. With this in mind, three types of end-user documentation are covered in this section 1) inline, 2) application help, and 3) software manuals. Inline documentation consists of all of the helpful information presented to the user during the execution of the current task. It includes, but is not limited to tool tips, descriptive labels, and context-sensitive help. This is arguably the most important documentation for the user and is typically the most underdeveloped. Proper inline documentation guides the user through each task, making it very clear what each element does, its intended use, and the expected values. It is not sufficient to explain how to use each control on a form or dialog box because this is akin to explaining how to fly a plane by describing each gauge. Enhancing the quality of inline documentation improves the application's usability making it less likely that the user will have to resort to additional documentation.

The application help is analogous to what users expect to see when they select "Help" in Microsoft Office. The application help becomes the user's next source of information when not enough information is provided by inline documentation. Consulting this help documentation interrupts the user's workflow and is usually caused by one of the following situations:

- The user needs an overview of what the application does and how to do it
- The user is executing a task and needs help to complete it
- The user needs to perform a specific task, but does not know how to begin
- The user wants to fix something that appears to be broken, or find an explanation for its behavior

Developers should keep in mind that the application help is not designed for them; that is, the documentation is intended for someone who is not familiar with the system. Developers often take for granted that steps such as right clicking on a blank window to proceed might not be obvious to someone who did not design the system. Within the application help, it is beneficial to list an application overview and some of the most common tasks as the initial listing of help categories. Additionally, common errors or a troubleshooting section will be useful to those users trying to understand why an error or anomalous behavior appears.

Software manuals have fallen out of use for two reasons. First, printing manuals is a costly process. Second, users do not read the manuals. Some users do not even have their manuals anymore. Digital and online manuals can offset those concerns, but a manual in digital form is just as accessible as application help, so if the user could not find enough information in the application help, they will not be likely to look in the online manual.

Manuals are an excellent place to put reference information for the application. The manual is the best place to put expanded technical and background information about the application that does not belong in the task-oriented application help. Additional overview and tutorial material are also useful in a software manual. If the application requires installation or configuration of client components, the relevant information must be published in the manual because the user cannot reach any other help until the application is available.

4.3.2 Internal Team Documentation

Documentation is not limited to help files and manuals written for the user. It also includes documents to assist the development team with daily activities. The specific requirements for development documentation will vary by organization, but there are some important forms of documentation that each team should consider having regardless of organizational requirement. As with most internal documentation, these elements may seem to take up valuable time, but following them properly can actually improve efficiency and quality by preventing time consuming mistakes.

4.3.2.1 Development Log

The development log is an extremely useful tool for tracking and coordinating the development team's efforts. This does not mean that its use is intended as a management activity; rather, its goal is to be a document that captures the institutional memory of the design and development progress. Another way to view this document is as the reference to consult when questions arise in a meeting concerning past events or design decisions. The development log should not serve as a detailed record of all of the changes to the source code because the Software Configuration Management tool should contain all of that information within the comments for each submission.

As mentioned earlier, some development tracking information should be included, i.e., a high-level view of what has already been accomplished, the next five steps to accomplish, and the upcoming focus areas. In addition to following the high-level development process, the log should contain the overall architecture of the application, including the decision process used to arrive at that architecture or the reasoning behind its selection. A simple statement such as "I created the architecture from my own knowledge" is sufficient, just as long as the origins are documented. Keep in mind that obtaining architectures from fortune cookies may lead to potential outsourcing. In this case, the value of the development log is in demonstrating the origins of a design by demonstrating the decision-making process and clearly showing they did not emerge from a fortune cookie.

In addition to the architecture, the development log should contain information tracking the division of labor for the development efforts. This allows questions about specific components to be directed to the expert on that portion of the source code. The benefit is in reducing the social overhead of asking around. On a small enough team, this may not be an issue, but the log is still a beneficial way of tracking the life history of components when the developers working on them change.

The log should also record the tricky problems the team encounters and how they were solved. This is especially useful if a similar problem arises in later development after the original development team has moved on to different projects. It may not be possible to contact the original developers, but the solution captured in the log may be reapplied, saving time and resources.

4.3.2.2 Source Code Comments

Writing source code comments is similar to eating vegetables—it has been long established that it is good for you, but that does not make it any more appetizing. Source code comments are arguably the most essential form of internal team documentation. Well documented source code allows anyone examining the code to understand its behavior without tracing through all of the steps and iterations within the code. Despite this benefit, the practice of commenting source code has fallen into disuse. Opponents to commenting source believe the source code should be self-documenting. This rationalization stems from concern about comment synchronization—the idea that source code is always current, whereas comments written for that code may not keep up with the code revisions. While comment synchronization problems are a possibility, the main reason developers dislike commenting code is due to a past history of poorly written comments. Poorly written comments seem tedious and useless and are perceived as a waste of time.

The fundamental question to ask when evaluating comments is: "Does this comment exist for a reason?" If the answer is not a resounding "YES," then the comment is unnecessary and should not be used. Most poorly written comments do not satisfy this basic test. Two general types of comments fall into this category—noise comments and formal documentation blocks.

Noise comments are the natural byproduct of forcing developers to generate comments as a matter of policy. This typically first occurs during programming coursework where the instructor dictates that all source code must be commented. This commenting style strives for quantity over quality and is possibly what causes developers to dismiss commenting as mere "busy work." This type of commenting rarely has any use; it is so finely granular that it is no more effective than simply reading the source code. See Figure 4.3.2.2 A for an example of code containing noise comments.

```
//addSweetness function
public void addSweetness(int nQuantity, int nType, Coffee currentCup)
{
   //switch on sweetner type
   switch(nType)
   {
   case 0:
      //iterate through each sweetner
      for (int i = 1; i < nQuantity; i++)
         currentCup.add(SweenterFactory.getEqual());
      break;
   case 1:
      //iterate through each sweetner
      for (int i = 1; i < nQuantity; i++)
         currentCup.add(SweetnerFactory.getSplenda());
      break;
   case 2:
      //iterate through each sweetner
      for (int i = 1; i < nQuantity; i++)
         currentCup.add(SweetnerFactory.getSugar());
      break;
   }
   //End switch-case block
}
```

Figure 4.3.2.2 A: Code Sample with Noise Comments

Formal documentation blocks represent another extreme—it attempts to solve the documentation problem by imposing a rigid documentation block for each function, class, etc. This comment style also discourages commenting source code because of its rigidity and is seemingly pointless because much of the information within the block can be gleaned from other sources, even if neither an SCCS system nor a defect tracker is available. In the following example, elements such as the procedure name, class name, inputs, and outputs can be determined by examining the source code. Comments tracking the author and revision history should be contained within an SCCS solution. The intent here is not to dismiss all formal comment blocks as bad commenting style, but merely to show how other styles may be more effective. See Figure 4.3.2.2 B for an example of code using a formal block.

```
//Procedure: addSweetness
//Class: Barista
//Filename: Barista.java
//Author: Joe D. Programmer
//Creation Date: March 15, 2004
//
// Inputs:
//    nQuantity: the number of sweetener packets to add
//    nType: the type of sweetener to use
//           0 – Equal
//           1 – Splenda
//           2 – Sugar
// Outputs:
//    NONE
//
// Revision History:
//    15 March 2004: Initial Creation
//    08 April 2004: Added Equal sweetner option
//

public void addSweetness(int nQuantity, int nType, Coffee currentCup)
{
   .
   .   //Code removed for sake of example
   .
}
```

Figure 4.3.2.2 B: Code Sample with Formal Block

One type of comment that is typically very useful is the placeholder comment. These time-traveling comments are placed in the code as future reminders for the developer. Some examples are shown in Figure 4.3.2.2 C.

```
// TODO: Replace this algorithm with a slightly faster version O(n!)
vs. O(n^2)

// HACK: Quick-fix to out-of-bounds problem, replace with proper error
checking

// POSTPONE: We’ll want to add a custom version of this dialog box
later
```

Figure 4.3.2.2 C: Code Sample with Placeholder Comments

Although these types of comments pass the test of having a specific purpose, they are better recorded elsewhere to avoid sifting through the source code to find all of the reminders. Some tools manage placeholder comments. If such tools are available, then these types of comments can be properly managed. However, if such a tool is not used, these types of entries should be entered into the defect tracking system as feature requests instead of actual defects (unless one of the notes indicates a case where a defect would occur). For this approach to be successful, the development environment should not fixate on entries within the defect tracker. Again, it is good practice to use a defect tracker as part of a process instead of as a process; otherwise, the developers will not place these useful comments into the system. Excessive criticism or focus on entries within the bug tracking comments may result a lack of developer candor.

With all of this build-up, it is useful to demonstrate some beneficial commenting styles. The two most useful forms of source code comments are summary comments, often called block comments, and intent comments. Summary comments are a vast improvement over the noise comments in that they are still compact, but actually have a reason for existing. Summary comments contain a quick summary of the functionality of the immediately following code block. This commenting style is intended to produce comments that summarize the code so actually reading the code is not required; resulting in less time spent analyzing the code to understand its function. In fact, a well-written summary comment should contain information that could be readily gleaned from reading the code. So, why should someone write summary comments? Simply put, even if the summary is a direct recap of the code, it is still shorter than reading and analyzing the code itself. Also, many editors color-code comment blocks, so it would be possible for someone to skim through all of the summary comment blocks and understand the flow of the source code. Figure 4.3.2.2 D shows an example of code containing a summary content.

```
public void addSweetness(int nQuantity, int nType, Coffee currentCup)
{
   //Depending on the type of sweetener selected (nType)
   //get nQuantity packets individually and add them into
   //the current cup
   switch(nType)
   {
      .
      .      //Again, removed for example
      .
   }
}
```

Figure 4.3.2.2 D: Code Sample with Summary Comment

Intent comments are arguably the most difficult type of source code comments to write. However, they also result in the greatest payoff. Intent comments literally explain the intent behind the existence of a specific code block. In formal comment blocks, this is usually encompassed by a "Description" entry in the block and is one of the most useful components of such a block. As opposed to summary comments that explain what a particular block of code *does*, intent comments explain *why* a particular block of code exists. Think of intent comments as an "elevator speech" for why that code belongs in the program at all. With quality intent comments, it is possible to very quickly go through code to find functional pieces based on the intent. Additionally, quality comments allow the developer to return to the same mental frame of reference when updating a code segment. See Figure 4.3.2.2 E for an example of code containing intent comments.

```
//The addSweetness function takes a Coffee object and adds sweetener
//objects to it until the maximum number of nQuantity has been
//reached. addSweetness is a separate helper function for controlling
//this specific element of preparing the Coffee
public void addSweetness(int nQuantity, int nType, Coffee currentCup)
{
   .
   .   //Removed function body for example
   .
}
```

Figure 4.3.2.2 E: Code Sample with an Intent Comment

4.3.3 Application Program Interface (API) and Reference Documentation

Somewhere between the highly technical detail present in internal team documentation and the task-based documentation for the end user lies reference documentation. This type of documentation is required of any team that develops software components for other teams' use and consists of all the information needed by developers to interact with a specific interface. Typically, this consists of API documentation with additional explanation. To clarify the model of interaction, imagine that a development team has been given a small black box with a collection of plugs and switches on it and that this team has been instructed to use this box in their development because it does a lot of work for them. Since the box is black, the development team cannot see inside to figure out what each switch does, nor would they be able to figure out what is and is not supposed to go into each plug. This leaves the team a few options:

1. Consult the reference documentation (if it exists)
2. Reverse engineer the interface using trial and error to find out what "works"
3. Sink the black box offshore for an artificial reef (so the box actually has some use)

The importance of the reference documentation is that it allows other development teams to leverage previous work and avoids "re-inventing the wheel". If the reference documentation is missing or inadequate, option two is usually taken. The danger of allowing this to happen is that integration among components occurs on a trial-and-error basis, injects errors for later discovery in system or integration testing, and results in finger pointing when those errors are discovered. Option three defeats the purpose of releasing a component for general use. If a development team has to throw away a component and "roll their own," then the effort is wasted due to a lack of proper documentation. This also presents a potential problem—a development team that writes their own version of a component will implement it differently than the original. This may cause interoperability problems or bugs due to the new interpretation of the component's intended functionality. Then again, sometimes this results in a higher quality component with a greater feature set and better documentation than the original version.

Because of these possibilities, the third option should be thoroughly considered prior to selection.

Well written summary and intent comments, as described earlier, greatly ease the development of reference documentation. They provide the context and understanding missing from a simple listing of all of the available functions or operations. Thus, at a minimum, good reference documentation includes all of the available functions and operations (the APIs), summary information to concisely describe what the function does, and intent information describing the intended use of each function.

There are some automated tools, such as JavaDoc for the Java environment, which allow the automatic generation of API documentation directly from the source code. This allows developers to leverage the existing source code documentation as a starting point for reference documentation. Developers should avoid the trap of believing that copying and pasting the comments from the source code will result in useful reference documentation. Frequently, summary and intent comments within the source code assume a baseline familiarity with the inner-workings of the component, knowledge an external developer would not have. To create truly useful documentation, additional context and organization needs to be added to the original source code comments to provide an overall view of not only how each function or operation works, but also how it all fits together. Likewise, when the operations are web service operations, it is not sufficient to claim the documentation consists of all of the internal annotations to the relevant XML schemas and WSDLs. These annotations are akin to source code comments and should likewise be organized into a reference that is greater than the sum of its parts.

4.4 Development Techniques

The following techniques individually have benefits to the overall development process and represent a set of small, simple changes that result in an overall improvement in the quality or flexibility of the software product.

4.4.1 Configuration Files

Configuration files provide the software application developer and system administrator the ability to make changes to the behavior of the code without having to make updates to the software. Configuration files are analogous to the user's ability to set preferences within an application, but provide the developer the flexibility to set variables for the software outside of the source code.

Any static variable within the code makes a good candidate for being contained in a configuration file. One example is the endpoints for services used by a web service application. A string such as "http://www.microsoft.com/axis/HLFreezeOver" can be placed in the configuration file and updated if the application gets hosted in another location or if the endpoint for the service changes locations. In addition to static variables, Boolean variables can be placed into the configuration files to set options to be toggled from outside of the code.

The Java programming environment provides a configuration file capability known as "Properties Files". Within the Java libraries, the Properties class can open a file with values stored per Figure 4.4.1A.

```
propertyName=property value
```

Figure 4.4.1A: propertyName Value

An example of property file content is shown below, followed by the Java source code to extract the values. The example code is minimized for demonstrative purposes; therefore, it does not contain the error checking or exception handling that it would in the original source. In Figure 4.4.1 B and Figure 4.4.1 C, lines beginning with a # indicate comments.

```
# Web Service endpoint for Security Adjudication Service
cpds_endpoint=http://wstest.jcdx.org/axis/services/SecurityAdjudication

# Provider ID number for Federated Search
providerID=01-01-2358

# Optional Processing Note
processingNote=THIS IS TEST DATA. ALL DATA IS UNCLASSIFIED.

# Print stack traces when exceptions thrown
printTrace=true
```

Figure 4.4.1 B: Example Properties File Entries

```
Properties prop = new Properties();

URL url =
CoffeeRun.class.getClassLoader().getResource("CoffeeRun.properties");

prop.load(url.openStream());

printTrace = java.lang.Boolean.valueOf(prop.getProperty("printTrace",
"true")).booleanValue();

strProcessingNote = prop.getProperty("processingNote", "");

cPDSEndpoint = prop.getProperty("cpds_endpoint", "");

strProviderID = prop.getProperty("providerID", "");
```

Figure 4.4.1 C: Example Source Code for Processing Properties File

To understand more about using the Properties class, consult the Java documentation.

4.4.2 Logging

Logging tools provide a level of visibility into the functioning of the code not normally present on the system without the use of a debugger. In distributed computing environments, such as an SOA, the use of a true debugger to step through code becomes difficult if not impossible; therefore, logging becomes one of the only tools available to the developer to check variable values, code execution paths, and perform sanity checks.

Although it is possible to perform logging to the console using standard output (System.out in Java; cout/cerr in C++, etc.), this is not the recommended method for logging because of the load that it places on the code. For example, if standard output logging statements are used to debug the code, they have to be removed prior to releasing the code, but this method does not provide a means to debug any release version of the code when problems show. In contrast, advanced logging packages allow developers to establish a hierarchical order of logging. This hierarchical model allows the developer to configure which logging level should be used when the code runs via a configuration file. Because of this, developers can insert all of the logging statements needed for debugging the code, including errors, warnings, etc., and filter out all of the debugging information when the code ships, but have the ability to obtain the log entries in the higher categories, such as errors.

Advanced logging tools also allow the developer to direct the logging output to alternative devices, so all messages do not have to go to the console. For example, a developer might put all debugging level statements into the console but trap errors in a special file. This ability allows logging packages to be used for auditing transactions carried out by the code.

4.4.3 Code Refactoring

Code refactoring is a technique publicized by eXtreme Programming (XP) advocates and refers to the process of changing software to improve the internal structure of the code without changing any of its external behavior. Code refactoring produces "cleaner" code that is usually more efficient and easier for other developers on the team to read and understand.

Refactoring has a set of common patterns. One of these is taking repetitive blocks of code and replacing them with calls to a single function that contains the repetitive content. Because of the repetitive nature of the patterns, code refactoring lends itself well to using automated tools, but also presents some level of risk. Code refactoring is usually done as a part of test-driven development; therefore, in this case, the unit tests already exist to catch any mistakes made during the refactoring process, thereby reducing the risk in altering the internal structure of the code.

4.5 Usability Tips

Usability is an often overlooked area of software improvement. The reason for this is that small usability issues seem minor relative to the rest of an application. Although usability seems like a minor issue, designing applications with it in mind accomplishes two things. First, a very usable application is inherently intuitive to use, minimizes training requirements, and minimizes the need for users to consult help documentation. Second, small usability issues individually contribute to general user frustration. The latter point may seem a minor one; however, as the user's frustration level increases the perceived quality of the software declines. The real trouble with this is that the perception is an intuitive feel and most users will not be able to explain why the software is "crappy". If they do try to explain, they will do so by pointing out the small frustrations caused by a lack of usability. The former point is an underemphasized one as well, but the best way to remember that point is the following statement:

> *"If my guys can't figure out how to use it in an hour, it's worthless to me"*
> *- MAJ Kurtis Warner, US Army, XVIII Airborne Corp*

Know your audience. As a developer, it is important to keep the operators' perspective in mind during development and testing to ensure your software will be easier and more efficient than the tools on which they currently depend. One way to examine the software from the user's perspective involves creating model users for the developers to use as a target audience. For example, Joe Smith is the model user for Common Operating Picture (COP) software. His profile includes general Windows platform computing knowledge and his daily tasking requires quick updates to the COP over Iraq. Since Joe Smith has general computing knowledge for daily tasks, it would be fair to assume he is familiar with products such as Microsoft Word and is comfortable with their interfaces and methods. With this knowledge, the developer would not build the type interfaces commonly found on a UNIX platform, instead focusing on developing the interface to be similar to those with which the user is already familiar.

Run tests to understand user expectations. Ask personnel who are not familiar with the application to use it while observing their actions. Focus on the steps they try to take and the amount of time needed to complete a task. These observations will give key insight into where the application could be improved for ease of use and efficiency improvements. The following example demonstrates a user testing a logistics tracking application:

> *On an Army logistics application, a decision was made during development to default the menus to all of the available logistics areas because the developers did not want to limit the choices available to the user. During user testing, it was observed that each user had a specific area of logistics with which they were primarily concerned. They*

> *expressed frustration at having to drill-down through all of the available menu options to reach their focus area and expected that their primary focus would be configurable as a user preference. The developers incorporated this feedback into the application; allowing users to set their primary focus area as a preference tied to their account. By allowing users to customize the default view based on their login, the number of steps required to fulfill their task was reduced, thus increasing their job efficiency and reducing their frustration with the application.*

Give fewer options. During construction, developers are frequently faced with design decisions about how the application should behave. Instead of analyzing user expectations, the developers elect to make the design choice a user-configurable option. This should only be done in cases where the option affords beneficial flexibility in the application functionality. Otherwise, this leads to an "overabundance of choice," which has two detrimental effects on the user. First, configuring options adds to the steps the user needs to execute during a task. Second, having too many available options creates "option paralysis" where the user is not sure which option to select. For example, most users do not care about dragging their toolbar all over the screen. In most, if not all cases, this just is not necessary, so the toolbar can be displayed at the top of the window. This also avoids the "What happened?" when the user accidentally uses an option they did not know existed—like moving the toolbar off the screen.

Give status notifications. Status notifications are helpful when users are stepping through tasks which might require them to use multiple forms, mouse clicks, etc. For example, a status indicator like "Currently Processing Submission" gives the user confidence that they are proceeding correctly. This also means that there should not be any indication that a process is complete before it is actually complete. For example, if a user submits a form that returns a hypertext link, the link should not be returned until the form is completely finished being processed. By nature, users take the link as notification that their request has been processed, so they will click on the link. If the link must be returned, it should be disabled or otherwise noted as unusable until processing is complete.

Be consistent with presentation. When different teams work on separate elements of the user interface, it is easy to finish with completely different looking interfaces. Keeping the appearances of these common interfaces consistent will reduce the learning curve for the applications, which users always appreciate. Users will adapt better to the software if they do not have the frustration of figuring out how to operate applications with vastly different user interfaces. For example, in a portal environment, the "help" icon should be consistent throughout the environment. This way, users looking for help have a consistent icon to direct their focus. This is especially important for the application help, as users may already feel frustrated by the time they look for the help and should not be subjected to a "help scavenger hunt".

Use proper icons and text. Use icons and text that lead the user in the intended direction. For example, only use button elements when the intent is for the user to click the items. If a button is for appearances only, users will assume the software does not work properly if nothing happens when they click on it. Similarly, do not make items appear like a hypertext link if the intent is for them not to be clicked. On the other hand, if the intent is for a user to click on a hypertext link, make sure that it looks like a link. Do not use standard text and expect the user to find the link.

Write concisely. Most users do not take the time to read long dialog boxes or directions on web pages. For example, the instructions might read: "Please choose one of the options below then click the submit button to have your information processed." If your instruction reads, "Choose an option then hit submit", more users would follow the directions because they are concise and easily understood.

Use descriptive labels. Tooltips and descriptive labels help the user understand the intended content without having to wait for validation. These types of labels also help users understand the function of an application without having to consult the application help.

Write friendly error messages. Error messages should be concise and understandable. Developers should not leave highly detailed technical debugging information in error messages of operational software. This type of information should be stored in an error log with the user-facing error containing a number indicating the corresponding entry in the error log. Providing a descriptive and friendly error message offers the user some assistance in figuring out what is wrong, which can aid in troubleshooting.

Reduce graphics file size. By reducing graphics file sizes in a web application, either through compression or physical dimensions, page load times can be decreased. The decrease in page load times make the user think that the application is more responsive, and thus, easier to use.

Adhere to standards. Web layouts do not always render the same on different browsers, such as Netscape Navigator and Internet Explorer. This is typically because layouts will be designed for a specific browser. Rather, develop layouts that adhere to industry standards, then make accommodations for the most popular implementations' nuances (i.e., develop a HTML 4.01 web page, then tweak the formatting so that it looks good in Internet Explorer, as opposed to developing the page using Internet Explorer).

Use case-insensitive URLs. Although host names are case-insensitive, path names are not. For example, http://www.home.com is equivalent to http://WWW.HOME.COM; thus, the expectation is that http://www.home.com/axisportal is equivalent to http://www.home.com/axisPortal. Case-insensitivity is consistent with general user expectations and using case-insensitive URLs prevent problems when users write down URLs that do not match the case-sensitive version.

Design application flow to match reading flow. Remember, most western languages read Left-to-Right; Top-to-Bottom. Forms should be laid out so they flow in the same direction that users read. This leads to a more natural-feeling flow because the element-to-element progression is intuitive.

4.5.1 Additional References

For further reading, the following websites discuss usability and were referenced for the previous tips.

- http://www.useit.com/
- http://www.usabilityfirst.com/
- http://www.microsoft.com/usability/publications.htm
- http://www.joelonsoftware.com/printerFriendly/uibook/fog0000000249.html

5 SECURITY AND DISCRETIONARY ACCESS CONTROL-PLUS (DAC+)

One of the key goals of Horizontal Fusion was to demonstrate secure communications (wireless or otherwise) and cross-domain information exchange from within an SOA. To achieve these goals, Horizontal Fusion targeted to deliver an SOA that met Protection Level (PL)-5 requirements following Director of Central Intelligence Directive (DCID) 6/3 guidelines. DCID 6/3 and security policies in general were not developed to address an SOA and, therefore, did not cleanly apply to the Horizontal Fusion effort (trusted operating systems vs. security services architecture). The Horizontal Fusion Portfolio did manage to achieve DCID 6/3 PL-3 requirements by moving beyond DAC towards Mandatory Access Control (MAC) requirements. The Portfolio labeled this effort as Discretionary Access Control-Plus (DAC+), where the plus represented those aspects of MAC that were implemented. In summary, these requirements included the following:

1. All personnel, servers, and mobile code signers were issued a PKI certificate as a key component of the architecture. The certificates were used by the infrastructure (portal, back-end services, etc.) as the identity token to provide client and server authentication when accessing resources, as well as the digital signature and encryption for SOAP messages, SAML assertions, labeling of data, and whatever mobile components traverse the enterprise.
2. All data in the enterprise, whether in motion or at rest, was labeled with a metadata tag which indicated classification of data. Labeling in this sense refers to the binding of the clearance information to the data elements in such a manner that it cannot be illegitimately changed.
3. Back-end services performed auditing of a defined set of events (e.g., service access failures, classification changes, etc.).

Policy decisions (data and service access) were made based on role, clearance, and citizenship user attributes in the LDAP directory (that were tied to the identity token/PKI certificate) using the security services.

Security has played an important role in the Horizontal Fusion Portfolio. Major achievements have included the addition of DAC+. DAC+ is the set of standard security control mechanisms and protocol implementations including SSL for HTTP, PKI authentication, Role-Based Access Control (RBAC), clearance based access, data labeling, controlled interface, and auditing.

DAC+ security additions should be configured in such a way that each DAC+ addition can be enabled or disabled based on a property file or startup configuration. Because the requirements for Horizontal Fusion are leading edge and very often ahead of the current technology baseline, not all of the DAC+ features can be implemented operationally at this time; however, the goal is to continue leading edge technology and innovation. Through properties that can be modified easily and without code change, DAC+ features should be configurable in this manner.

5.1 Secure Socket Layer (SSL)

SSL plays an important role in communication both between client and server and server to server. Server Certificates enable strong encryption for connections made. Likewise, PKI solutions add an additional layer for client to server authentication. Two-way SSL requires both the recipient of the request to provide a certificate and the sender to authenticate. Also, applications that connect to a server must utilize two-way SSL, providing a client certificate to the server as the server provides one to the application. This form of encrypted communication is more secure.

5.2 Public Key Infrastructure (PKI)

PKI is the use of a Personal Identity Certificate, often referred to as a PKI Certificate, to enable encryption and identification for authentication and secure communication. PKI also serves the purpose of digitally signing electronic information. In Horizontal Fusion, PKI certificates are issued for both servers and users. When passing certificate information between services, clients, servers, services should include the full Distinguished Name (D/N) of a certificate where possible.

5.2.1 Server Certificates

Server Certificates need to be obtained and configured on any server that is going to digitally sign SOAP messages and SAML assertions. Also, server certificates enable two-way SSL that will be discussed in a later section. Server Certificates are different from personal certificates because they identify a server and not an individual user. Server Certificates should reflect a fully qualified domain name of the server that will utilize this certificate.

The process for obtaining a server certificate is highly dependant on the physical location of the server and which network it is based on. Contact your Local Registration Authority (LRA) to obtain the proper forms and security packages before starting the certificate process. A few tips to remember when applying for a server certificate:

- Update all operating system software and install the latest security patches
- Virus software is required
- Passwords cannot be stored in plain text

- Written justification of all compilers, interpreters, and code-based applications is required

5.2.2 Personal Certificates

Personal Certificates are issued to each individual and solely to that individual. These certificates have an associated D/N, such as ou=PKI, ou=DoD, o=U.S. Government, c=US and a Certified Name (C/N) which normally consists of LastName.FirstName.Mi.03949394 where the Last Name, First Name, and Middle Initial are followed by a unique number. Combined these make a full entry in LDAP that can have associated values like role, clearance, and citizenship.

The processes for obtaining a DoD and an External Certificate Authority (ECA) PKI certificate are different. Factors include your employment status, employer, and whether the PKI is for the Non-Secure Internet Protocol Router Network (NIPRNet) or the SECRET Internet Protocol Router Network (SIPRNet). All DoD employees (civilian and military) including contractors using Government Furnished Equipment (GFE) are eligible for DoD PKI certificates. Contractors supporting DoD and other U.S. Government agencies and who are not using GFE are eligible for ECA PKI certificates which are NIPRNet only. If you are a DoD employee or DoD contractor using GFE, the particular steps for obtaining a NIPRNet or a SIPRNet PKI certificate are specific to your local LRA.
A few things to remember with your PKI certificate:

- DoD employees and contractors using GFE will need two forms of government issued picture identification to obtain both a NIPRNet and a SIPRNet PKI. If you do not have two forms of picture identification, you must obtain either a Passport or an additional government issued picture identification card. Plan for this requirement.
- All Contractor employees will need the same information as DoD Employees to obtain a SIPRNet PKI.
- Contractor employees, not using GFE, requesting a NIPRNet PKI, must use one of the organizations authorized to be an ECA. The ECAs require three forms of identification, one picture and two others with your name (i.e., major credit card, insurance, copy of utility bill). The applications must be notarized.
- NIPRNet and SIPRNet PKIs can take up to four weeks to be issued. Plan accordingly.

5.2.3 Code Signing

All mobile code should be signed before distribution. Horizontal Fusion has a process to digitally sign .cab and .jar files for distribution. These files represent the compressed libraries that are downloaded by a client and run client side, such as ActiveX components and Java Applets.

5.2.4 Trust Stores and Root Authorities

Trust Stores are files that contain the trusted elements of a certificate. A server need not trust every certificate manually but merely trust the issuer of the certificate. Trust Stores are very important in the process of encryption and digital signing, as some level of trust must be maintained. Server administrators are encouraged to pay attention to the Root Certificate Authorities (CA) that Horizontal Fusion uses and include only explicitly specified CAs and Root CAs. You can find public root authority stores for DoD and ECA on the web.

5.3 Attribute-Based Access Control

5.3.1 Role-Based Access Control (RBAC)

RBAC is the use of role policies to protect and govern data and applications. PDS is a core service that helps services control access based on a user’s roles. Assume a service contains several operations to manipulate a database called getResult, addValues, updateValues, and deleteRecord. Assume also that a user can have the role of user or admin. In this highly simplified example, only users with the admin role should be able to delete and add values; however, any user is allowed to read and update values and as such a user with the user role or admin role could read the data. RBAC policy should be implemented in such a way to allow access to those with the required roles. For example, the code should check to see that a user has either the user role or admin role before allowing access to read the data from the database.

When writing RBAC enabled code, remember that role schemas may change and thus hard coding roles is not a valid solution. PDS stores role policy and should thus be used to maintain accurate role policy. Not only is this method more general and extensible, but also this reduces the overall coding effort. PDS will make the decision based on the role policy entered into the PDS administration server that is taken care of during service registration with UDDI. Role information of a user is stored in LDAP along with other information, such as name and clearance.

PDS utilizes a service WSDL document and manual role policy to set the role access set. However, PDS cannot be used for configuring access to data. Some data may only be available to users with specific roles. In which case, some mechanism must be designed and implemented within the code to check a user’s credentials. For FY2004, Horizontal Fusion used the following role schema:

- user.analyst
- user.analyst.weather
- user.operator

- admin.system
- admin.database

For the most part, every user is assigned default roles of user.analyst and user.operator. For administrative type functions, persons specified will be given the admin roles. Likewise, user.analyst.weather is a special role created for Weather Specialists. When developing an RBAC enabled application, please remember:

- Do not hard-code roles
- Use Policy Decision Service
- Use Properties files to store role policy.

5.3.1.1 Role-Based Access Control (RBAC) Example

For the following example, there is a service called **TimeTempService**. TimeTempService has three operations getTime, getTemp, and sendLog. The purpose of the service and operations is irrelevant for this example; however, the getTime operation returns the time at specified latitude and longitude; the getTemp operation returns the temperature at a specified latitude and longitude; and the sendLogs operation returns the full error log to an administrator. The service has a Qname of `http://hf.spawar.navy.mil/TimeTempService#TimeTempService`. The role policy for this example is illustrated in Table 5.3.1.1 A.

Operation:	user.analyst	system.admin
Discover	YES	YES
getTime	YES	YES
getTemp	YES	YES
sendLogs	NO	YES

Table 5.3.1.1 A: Specific Role Policy Example

For security reasons, only an administrator or person with the system.admin role can view the logs of the service; however, any user with the user.analyst role can perform any other operation. The following users try to use the service. Each user is listed in Table 5.3.1.1 B with their D/N and roles.

User DN	**Roles**
Smith.Bob.01	user.analyst
Jones.Roger.03	user.operator, user.analyst
Doe.John.09	admin.system

Table 5.3.1.1 B: Example Users with DN and Roles

In Figure 5.3.1.1 A, Bob Smith tries to access the service and getTemp. Because he has the proper role, he is returned the time for the location he specified.

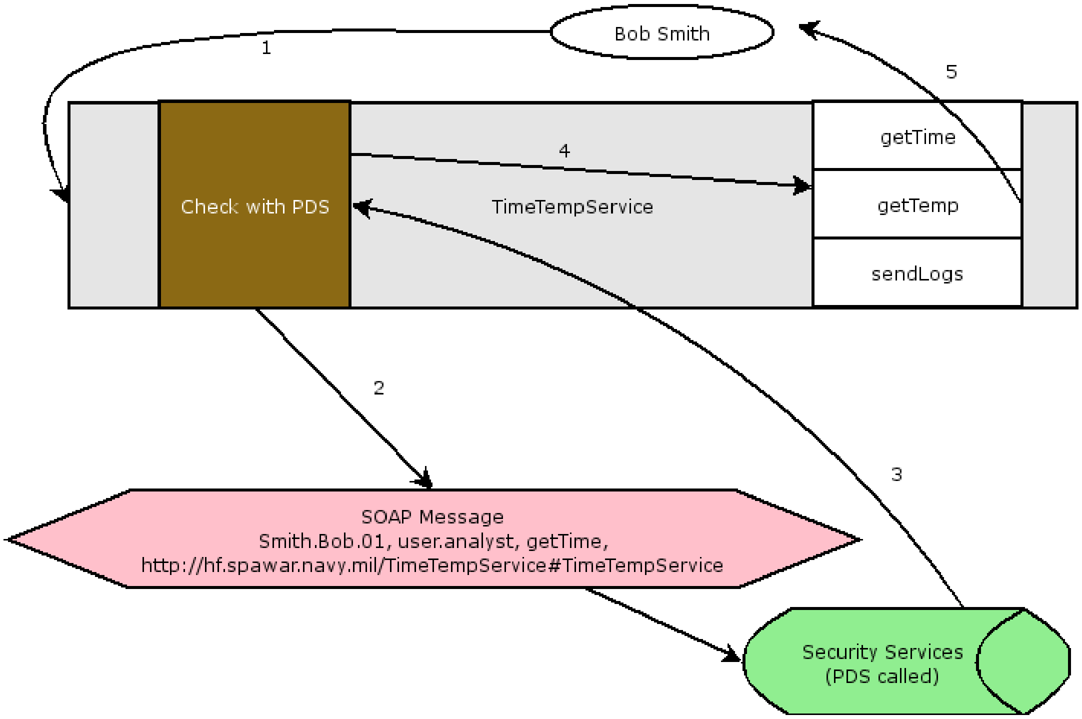


Figure 5.3.1.1 A: Illustration of User Accessing Services

Bob calls the TimeTempService (1). TimeTempService obtains Bob's D/N and Roles and then calls PDS with that information and the service Qname and operation bob is trying to call (2). Security Services, Policy Decision Service specifically, evaluates the SOAP message, extracts the information, and compares Bob's roles to the Role Policy. Because Bob has the user.analyst role, PDS returns a positive result (3). With the approval of PDS, TimeTempService calls getTemp (4) and returns the temperature to Bob (5).

Bob was successful. Roger Jones then tries to access the service. He wants to see what Bob was doing; however, Roger is not a system admin. He tries to call sendLogs. Figure 5.3.1.1 B illustrates his transaction.

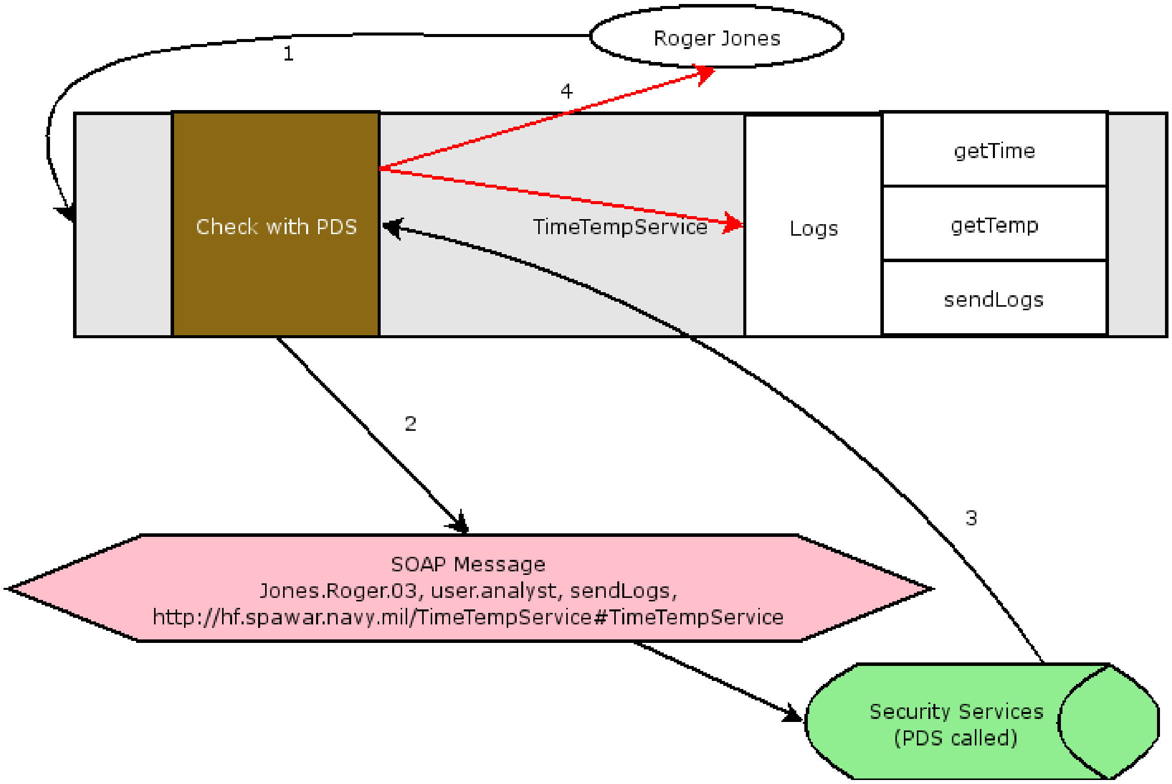


Figure 5.3.1.1 B: Transaction Illustration

Roger calls TimeTempService (1) trying to access sendLogs. TimeTempService calls PDS and sends all of Roger's information, as it did with Bob (2). However, PDS returns a negative response to the request (3). TimeTempService records all of the information about Roger and when he tried to call this service, including his IP address (4) and sends a fault message back to Roger.

These examples are a very basic model of how PDS can control simple RBAC. Developers should understand that only roles specified can have access and that there is no way to create the reverse. Developers must specify the roles that have access and not the roles that do not have access to each operation.

5.3.2 Clearance Based Access Control

Clearance-based access control is an extension to RBAC that recognizes that data and services may in fact be labeled with clearance levels. cPDS has been written and implemented much like the PDS service to handle the implementation of citizenship and clearance. SOAP Messages and SAML Assertions should be properly labeled and the user's clearance and citizenship should be taken into account before processing such access. Data should be labeled accordingly to its classification. That is to say, a document that has three paragraphs, two unclassified paragraphs and one SECRET paragraph, the document should be overall classified as SECRET.

5.4 Data Labeling and Marking

Data within the Horizontal Fusion SOA should be labeled. Unlike data marking, which simply applies proper metadata tags to the data itself, labeling involves the signing of data with the applied metadata markup in such a way that the metadata cannot be removed or modified without knowledge of this modification being detected by the end recipient of the data. Often proper data labeling involves signing a message or data with a server certificate, see the section on PKI. Data Labeling is important both at the transport layer and the storage layer. Proper labels should include classification and releasability markings. There are three major types of data labeling: visible classification, transport marking, and labeling at rest.

5.4.1 Visible Classification

All data that will be visibly readable or seen, such as text, documents, images, and web pages, should have visible classification. Labeling should adhere to the DoD Classification Labeling standards or standards set forth by the Horizontal Fusion Portfolio Management Team.

5.4.2 Transport Labeling

Data should also be labeled when sent. SOAP messages should contain metadata that contains classification and citizenship. Any visible data as mentioned in the Visible Classification section should still adhere to labeling requirements in addition to the metadata labeling.

As the data a service consumes can come from a variety of sources as well as be utilized by many sources, labeling plays an important role. Transport labeling is extremely important to allow other services to consume data and properly serve data based on classification and citizenship. Other services may utilize the data in other manners and as such require properly labeled data.

5.4.3 Labeling at Rest

Data may be stored for later retrieval. When data is stored it should be labeled appropriately. As the data may be consumed later, labeling data at rest allows for proper transport labeling as well as security measures.

5.5 Auditing

Auditing involves properly recording security-relevant information. Outbound messages should contain information such as identification (ID), timestamp of sending, target server, and IP recipient. Likewise, inbound messages should contain this same information and it should be logged in addition to IP of sender and timestamp of receipt. Success and failure of messages, security violations or failed authentication, access records including permit, deny, indeterminate, and service calls should all be logged. Logs should be readily available for inspection. Should logs be kept in XML format, a tool to view these logs should be readily available for non-technical persons. Logs should be restricted for review by authorized security personnel only.

Auditing and Logging are often confused or used interchangeably. However, the two are separate topics. Logging deals more with code output, malfunction, errors, and process. However, auditing is more for security-related events such as log-in, unauthorized attempts to access a service, failed log-in attempts, and other topics that hold merit within the realm of security and information assurance.

Auditing should record:

- Login (success, failure)
- Logout (success)
- Application and Session Initiation (failure)
- Use of Privileged Commands (failure)
- Use of Print Command (success)
- DAC Permission Modification (success, failure)
- Export to Media (success)
- Unauthorized attempts to Files (failure)
- System Startup/Shutdown (success, failure)
- Applying Classification (success, failure)
- Change of Classification metadata (success, failure)

Portlet Auditing should include:

- Login (success, failure)
- Logout (success)
- Application Initiation (failure)
- Unauthorized Attempts to Files (failure)

5.6 Controlled Interface

DCID 6/3 states: A *Controlled Interface* is a mechanism that facilitates adjudicating the security policies of different interconnected Information Systems (ISs) (e.g., controlling the flow of information into or out of an interconnected IS). When the user connects, the web server validates the user's credentials in LDAP, validates the user against PDS and cPDS for the appropriate service or data being requested, and ultimately returns the data or access. Controlled interface essentially touches every aspect of DAC+. As the communication between client and server and server to server must be SSL, a PKI must be presented for authentication; PDS and cPDS are utilized to validate the user's credentials; and all transactions are logged appropriately. Controlled interface does apply to static web pages. Authentication of the user is required even for static web content.

6 EXAMPLES

This section will provide brief examples of some of the technologies discussed in the previous sections. When evaluating these examples, remember that a complete and finished product is not the intent of these examples but a general foundation from which to start. These examples may serve as a starting point for developers.

6.1 Portlet Code Example

This section provides a very basic and simple Portlet including the portlet.xml file, myPortlet.java file, view.jsp, and Log4j configuration XML file. This is the sample Portlet that illustrates the most basic functions in a Portlet and some code that shows how to log using Log4j.

The portlet.xml file, shown in Figure 6.1 A, is what defines the Portlet. Using the schema provided by Sun Microsystems, the format for the portlet.xml file is strict. The Portlet tag is the root element of the XML document. The init-param tag specifies a initializing parameter that is read when the Portlet is initialized. Init-param values are essential providing information like URLs, role policies, paths, and other information that should not be hard coded into the Java.

```
 <?xml version="1.0" encoding="UTF-8"?>
 <portlet-app version="1.0"
  xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/portlet">

 <portlet>
  <description>myPortlet</description>
  <portlet-name>myPortlet</portlet-name>
   <display-name>Portlet</display-name>
   <portlet-class>myInitiative.myPortlet</portlet-class>

 <init-param>
   <name>jsppath</name>
  <value>/portlets/initiative</value>
 </init-param>

<init-param>
  <name>resourcepath</name>
  <value>/hfPortal/portlets/initiative</value>
 </init-param>

 <expiration-cache>-1</expiration-cache>
 <supports>
 <mime-type>text/html</mime-type>
<portlet-mode>VIEW</portlet-mode>
</supports>
```

```
 <supported-locale>en</supported-locale>
  <portlet-info>
    <title>My Portlet Name</title>
    <short-title>My Portlet Name</short-title>
    <keywords>My Portlet Name</keywords>
</portlet-info>
</portlet>

</portlet-app>
```

Figure 6.1 A: Portlet.xml

The myPortlet.java, shown in Figure 6.1 B, file represents the base class of the portlet and is specified in the portlet.xml file in the portlet-class tag. myPortlet.java is a very basic java file that retrieves the init parameters, sets up Log4j, and dispatches the view.jsp.

```
package myInitiative;

import javax.portlet.*;
import java.io.IOException;

public class myPortlet extends GenericPortlet {

    public void doView (RenderRequest request,
   RenderResponse response) throws PortletException, IOException
    {
        response.setContentType("text/html");

        String jsppath = getPortletConfig().getInitParameter("jsppath");

 private Logger logger = Logger.getLogger(myPortlet.class);

 logger.debug(" - in doView(), jsppath = " + jsppath);

        PortletRequestDispatcher rd =
getPortletContext().getRequestDispatcher(jsppath + "/view.jsp");

        rd.include(request,response);
    }
}
```

Figure 6.1 B: myPortlet.java

In the file above, the package name is identified and javax.portlet.* and java.io.IOException classes are imported. The Log4j logger is created and setup in Figure 6.1 C.

```
logger.debug(" - in doView(), jsppath = " + jsppath);
```

Figure 6.1 C: Log4j Logger Setup

Log4j, when set to include debug information, adds an entry informing developers an attempt to dispatch the view.jsp and includes the parameter values for debugging purposes.

The view.jsp, shown in Figure 6.1 D, file is parsed by the Java Application Server and the output is sent to the user. The user only sees the rendered portlet with “Hello World” and the image specified in the init-parms of the portlet.xml. The view.jsp page also contains debugging information for Log4j, in case the portlet does not render properly.

```
<%@ page session="false" %> <%@ page import="javax.portlet.*"%>
<%@ page import="java.util.*"%>
<%@ taglib uri="http://java.sun.com/portlet" prefix="portlet" %>
<portlet:defineObjects/>

<%
 Log logger = LogFactory.getLog(myPortlet.class);
 logger.debug(" - in view.jsp");

 PortletSession psession = renderRequest.getPortletSession();
 String path = portletConfig.getInitParameter("resourcepath");
%>

 <img src="<%= path + "/images/hello.gif" %>" align=middle>

 Hello World
```

Figure 6.1 D: view.jsp

The final file for this portlet is PortalLog4jConfig.xml file, shown in Figure 6.1 E, contains the configuration information for Log4j for this portlet. This file is generically the same for all portlets with modified paths and log names.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/"
debug="true">

  <!-- appender for myInitiative messages -->
 <appender name="myInitiative"
class="org.apache.log4j.RollingFileAppender">
        <param name="File" value="logs/myInitiative.log" />
        <param name="Append" value="true" />
        <param name="MaxFileSize" value="15000KB" />
  <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%d [%t] %-5p CLASS:%C
METHOD:%M LINE:%L - %m%n"/>
  </layout>
 </appender>

  <!-- categories for myInitiative messages -->
  <category name="com.myInitiative">
    <priority value="debug" />
    <appender-ref ref="myInitiative" />
  </category>

  <root>
```

```
    <priority value="fatal" />
    <appender-ref ref="SYSERRORLOGFILE"/>
  </root>

</log4j:configuration>
```

Figure 6.1 E: PortalLog4jConfig.xml

7 ACRONYMS

Acronym	Meaning
API	Application Program Interface
ASD/NII	Assistant Secretary of Defense for Networks and Information Integration
BTE	Bug/Defect Tracking Expert
C/N	Certified Name
CA	Certificate Authority
CES	Core Enterprise Services
COI	Community of Interest
COP	Common Operating Picture
cPDS	Classification Policy Decision Service
CVS	Concurrent Versioning System
D/N	Distinguished Name
DAC+	Discretionary Access Control-Plus
DCID	Director of Central Intelligence Directive
DISA	Defense Information Services Agency
DoD	Department of Defense
ECA	External Certificate Authority
FTP	File Transfer Protocol
GDS	Global Directory Service
GFE	Government Furnished Equipment
GIG-BE	Global Information Grid Bandwidth Expansion
GPL	General Public License
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol (Secure)
ID	Identification
IFIS	Intelligent Federated Index Search
IP	Internet Protocol
IT	Information Technology
J2EE	Java2 Platform, Enterprise Edition
JTRS	Joint Tactical Radio Systems
LDAP	Lightweight Directory Access Protocol
LRA	Local Registration Authority
NCES	Net-Centric Enterprise Services
NIPRNet	Non-Secure Internet Protocol Router Network
NSDS	NCES Service Discovery Service
OASIS	Organization for the Advancement of Structured Information Standards

Acronym	Meaning
ODBC	Open Database Connectivity
PDS	Policy Decision Service
PHP	HyperText PreProcessor
PKI	Public Key Infrastructure
QoS	Quality of Service
RBAC	Role-Based Access Control
RWS	Registration Web Service
SAML	Security Assertions Markup Language
SATCOM	Satellite Communications
SCCS	Source Code Control Systems
SDK	Software Development Kit
SIPRNet	SECRET Internet Protocol Router Network
SOA	Services-Oriented Architecture
SOAP	Simple Object Access Protocol
SSL	Secure Socket Layer
SWS	Search Web Service
TFG	Taxonomy Focus Group
tModel	Technical Model
UDDI	Universal Description, Discovery, and Integration
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WSDL	Web Service Descriptive Language
XACML	eXtensible Access Control Markup Language
XHTML	eXtensible HyperText Markup Language
XML	eXtensible Markup Language
XP	eXtreme Programming
XSD	XML Schema Definition